NANOLOG: A NANOSECOND SCALE LOGGING SYSTEM

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

STEPHEN YANG

MARCH 2020

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

John Ousterhout, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mendel Rosenblum

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Matei Zaharia

Approved for the Stanford University Committee on Graduate Studies.

Stacey F. Bent, Vice Provost for Graduate Education

Abstract

Instrumentation is fundamental to developing and maintaining applications in the modern datacenter. It affords visibility into what an application is doing at runtime, and it helps pin-point bugs in a system by exposing the steps that lead to an error. The most common method of instrumentation today is logging, or printing out human-readable messages during an application's execution. Unfortunately, as applications have evolved to become increasingly more performant with tighter latency requirements, traditional logging systems have not kept up. As a result, the cost of producing human-readable log messages is becoming prohibitively expensive.

NanoLog is a nanosecond scale logging system that's 1-2 orders of magnitude faster than existing logging systems such as Log4j2, spdlog, Boost log, or Event Tracing for Windows. The system achieves a throughput of up to 82 million log messages per second for simple log messages and has a typical log invocation overhead of 8 nanoseconds. For comparison, other modern logging applications today can only achieve up to a few million log messages per second at log latencies of hundreds of nanoseconds to several microseconds.

NanoLog achieves its ultra-low latency and high throughput by shifting work out of the runtime hot-path and into the compilation and post-execution phases of the application. More specifically, it performs compile-time extraction of static information from the log messages to reduce I/O and decouples formatting of the log messages from the runtime application by deferring it until after execution. The result is an optimized runtime that only outputs the minimal amount of data and produces a compact, binary log file. The binary log file is also amenable to log analytics engines; it is small relative to full, human-readable log messages and contains all the data in a binary format, saving the engine from parsing ASCII text. With these enhancements, NanoLog enables nanosecond scale logging and hopes to fill the performance gap left between traditional logging systems of today and the next generation applications of tomorrow.

Acknowledgments

When I first started the PhD program, I had no idea what the journey entailed or even how to get started. But thanks to a few key people in my life, I was able to navigate the landscape and graduate with this dissertation and degree in hand.

First and foremost, I would like to thank John Ousterhout for being my primary advisor for the past couple of years. He's guided me through becoming a better engineer, a better scientist, and more importantly, a better person. He's always supported my work, whether it be with him or other groups outside his lab. He's taught me how to analyze problems in ways that generalize beyond just research, and he'd always make time to meet when I needed[1]. He's allowed me to find and pursue my own hobbies and interests during the PhD program and even supported me for conferences outside his area of research. We'd often see eye-to-eye on many things in our meetings, but even if we don't, we'd often agree on how to disagree. When I am with John, I don't feel like I work *for* John, but rather I work *with* John. It's been a long journey, and I have learned a lot. Thanks John.

Also thank you to my committee members, Matei Zaharia, Mendel Rosenblum, Mac Schwager, and Dawson Engler for both providing feedback on my work and attending my thesis defense. Special thanks to Mendel Rosenblum for coming into our office every week and sharing insightful anecdotes from the industrial world. It helps the lab stay grounded and aware of the world outside our office's four walls (well, three regular walls and one glass wall).

Additionally, thank you Wendy Ju and Michael Bernstein for supporting my brief detour into the Mechanical Engineering Department. Thank you, Wendy for introducing me to the area of Human Robot Interaction and having the patience to deal with my lack of expertise in the new field. The work we've done together has taught me a lot and expanded my scope as a researcher. It's unfortunate that none of the research with you two made it into this dissertation, but I'm glad I was able

[1]And equally important, allowed meetings to be canceled when they weren't.

to help [14, 38, 39, 52, 53, 72] (Also, thank you for hosting the hot chocolate parties with the other graduate students at your house).

Thanks to my labmates, old and new, for their help and support in everything. Ryan Stutsman, you're an inspiration, and the stories of how you navigated your undergraduate education were awe-inspiring. Diego Ongaro, you're a cool dude. Steve Rumble, you are amazing with your coding skills, whether you realize it or not. Ankita Arvind Kejriwal, thanks for introducing me to your friends when I first arrived at Stanford, for inviting me to dinners (despite my voracious hunger for non-vegetarian food at the time), and for bringing me on to your research project; that last one was a SLIK move. Thank you, Henry Qin for all the *non-vegetarian* dinners and trips to Fremont, CA for good food. Thank you, Seo Jin Park for instrumenting Event Tracing for Windows and helping me pursue janky DIY projects. Collin Lee, thank you for the dinners, the cool stories, and guidance on actual engineering (unlike Super Jank Park's projects). Finally, thank you to Behnam Montazeri and Jonathan Ellithorpe for just being in lab everyday. It was nice to know that whenever I came into lab, there was always a friendly face around to greet me.

Thank you to my undergraduate friends who have made trips up to the Bay Area to visit me. Thank you Michael Wayne Li and David Dinh for inviting me on trips (and sometimes paying for them =p). Thank you, Adrián Galván for helping me move up to the Bay Area and helping me adjust to the new area. And thank you everyone else for being around when I come back to visit (you know who you are).

Thank you to my undergraduate research advisors at the University of California, Irvine for putting me on the PhD track. Thank you, Brian Demsky, for supporting my undergraduate research, including me in the Dynamic Out-of-Order Java work, and trying so hard to convince me to branch outside of UCI (it worked). Thank you, Tony Givargis, for providing support and guidance during my summer research and connecting me with other researchers.

Finally, thank *you*, fair reader, for starting this dissertation. It's a long one, but it represents the cumulation of everything I've learned about low-latency systems. In it, you'll not only find a description of NanoLog, but also tidbits of arcane knowledge like C++17 metaprogramming and CPU cache designs. I hope you enjoy.

Industrial Response To NanoLog

When I claim NanoLog can produce up to 82 million log messages per second with a median invocation latency of 7-19 nanoseconds, industrial affiliates typically respond with an expression similar to Figure 1[2].

[2]The image was created by user Sonitaaaaa on Wikimedia Commons and distributed under Creative Commons Attribution-Share Alike 4.0 International [56].

Figure 1: Caricature of the typical industrial response to NanoLog.

Contents

List of tables

List of figures

Chapter 1

Introduction

Today's datacenter applications are becoming increasingly disaggregated with ever tighter latency requirements. Driving the overall trend is user expectations and the rise of cloud computing; modern consumers want applications that are feature-rich, fault-tolerant, in the cloud, and fast. As a result, developers have broken traditionally monolithic system designs into collections of independent micro services that communicate with each other to provide higher-level functions. This drive increases the complexity of applications and reduces the latency tolerance for each individual component.

As a result, modern software systems are becoming harder to instrument with traditional tools. The disaggregated nature of applications means that one can no longer attach a debugger to a single process and step through the application[1]. The distributed environment can also induce "gray failures" where the overall performance is degraded, but does not cause a hard fault with an easy-to-debug stack trace [27], or there may be distributed, transient errors that are hard to observe with on-demand instrumentation or sampling. These issues are made worse because tighter latency requirements reduce the overall headroom one has to instrument the system. This leaves application developers with a limited set of tools to instrument their systems.

Fortunately, there is one form of instrumentation that remains both viable and popular in the modern, distributed world: printf-style logging [6]. Logging allows developers to persist portions of the program state onto disk or a database via log messages such as in Figure 1.1. This information can illuminate the steps that lead up to a crash in a microservice. The log can also contain metrics to

[1]Even if one could attach a debugger, the slowdown induced in the single process may cause other microservices to fail.

```
NANO_LOG(NOTICE, "Creating table '%s' with id %d", name, tableId);

2017/3/18 21:35:16.554575617 TableManager.cc:1031 NOTICE[4]: Creating table
'orders' with id 11
```

Figure 1.1: A typical logging statement (top) and the resulting output in the log file (bottom). "NOTICE" is a log severity level and "[4]" is a thread identifier.

help diagnose gray failures, as well as information about transient/recoverable errors or misconfigurations that lead to sub-optimal behaviors. Logging can also be always-on, meaning it will never miss an important event. Furthermore, logs can also persist user queries, behaviors, and preferences, which can be mined for business purposes. The more events that one can record in a log, the more valuable it becomes.

Unfortunately, logging today is becoming more expensive relative to modern applications. Simply formatting a log message takes on the order of one microsecond in typical logging systems. Additionally, each log message typically occupies 50-100 bytes, so available I/O bandwidth limits the rate at which log messages can be recorded. As a result, developers are often forced to make painful choices about which events to log; this impacts their ability to debug problems and understand system behavior.

The problem is exacerbated by the current trend towards low-latency applications and micro-services. Systems such as Redis [50], FaRM [9], MICA[33], and RAMCloud [45] can process requests in as little as 1-2 microseconds. This means that even a single log statement in these modern services can potentially double the application response time. This mismatch makes it difficult or impossible for companies to deploy low-latency services. One industry partner informed us that their company will not deploy low latency systems until there are logging systems fast enough to be used with them [13].

Slow logging is such a problem today that software development organizations find themselves spending valuable developer resources to remove log messages for the sake of performance. According to our contacts at Google[13] and VMware[44], a considerable amount of time is spent in code reviews discussing whether to keep log messages or remove them for performance. This process inadvertently culls a lot of useful debugging information, making the system harder to understand and resulting in many more person hours spent later debugging. Logging itself is expensive, but lacking proper logging is *very* expensive.

NanoLog is a new, open-source [70] logging system that attempts to address the performance limitations of traditional logging systems and close the gap between instrumentation and modern application latency requirements. NanoLog is 1-2 orders of magnitude faster than existing systems such as Log4j2 [62], spdlog [37], glog [21], Boost Log [1], or Event Tracing for Windows [47]. NanoLog retains the convenient printf-like API [6] of existing logging systems, but it offers a throughput of around 80 million messages per second for simple log messages and a caller latency of as little as 8 nanoseconds. For reference, Log4j2 only achieves a throughput of 1.5 million messages per second with latencies in the hundreds of nanoseconds for the same microbenchmark.

NanoLog achieves its ultra-low latency and high throughput by shifting work out of the runtime hot-path and into the compilation and post-execution phases of the application. More specifically, NanoLog introduces a new compile-time component that analyzes the log statements at compile-time to generate more optimized runtime functions, and NanoLog defers the formatting of log messages to a separate post-processor application. The addition of these two components shifts logging work out of the runtime hot-path, enabling nanosecond scale operations.

The compile-time component performs two types of optimizations on the log statements. For each log statement, it identifies the static information that never changes between log function invocations (such as the filename, line number, severity and format string in Figure 1.1) and puts it on a separate data path from the dynamically changing information (such as the timestamps and format arguments). The static information is stored just once in the log file, and the log statements are rewritten to persist the only dynamic information. This reduces the I/O requirements of logging, and overall reduces the amount of information that needs to be processed at runtime.

The post-processor component helps decouple formatting from runtime. Instead of fully formatting the human-readable log message as shown at the bottom of Figure 1.1, NanoLog instead emits only the raw components of the log messages at runtime. This results in a binary log file. A post-processor is then introduced that will read the components and format them. This completely removes the formatting costs from the runtime, which saves hundreds to thousands of nanoseconds per log statement.

The rest of this dissertation will motivate the techniques behind NanoLog, describe the various components of the system, evaluate the open-source implementation of NanoLog for C++ on GitHub [70], and conclude with extensions and future directions for the work.

Chapter 2

Background and Motivation

The NanoLog system builds upon techniques used by existing logging systems to achieve nanosecond scale logging. So to better situate the work in this space, I will discuss the printf-like API [6] that NanoLog inherits from existing logging systems, and describe some strategies that modern logging systems use to cope with the problem of slow logging. I will then conclude with two observations that NanoLog makes about modern day logging, and how those motivated the full design of the NanoLog system.

2.1 The API

Logging systems allow developers to generate a human-readable trace of an application during its execution. Most logging systems provide facilities similar to those in Figure 1.1. The developer annotates system code with logging statements. Each logging statement uses a printf-like interface[6] to specify a static string indicating what just happened and also some runtime data associated with the event. The logging system then adds supplemental information such as the time when the event occurred, the source code file and line number of the logging statement, a severity level, and the identifier of the logging thread.

2.2 Techniques to Speed Up Logging

The simplest implementation of a logging system is to output each log message synchronously, inline with the execution of the application. This approach has relatively low performance, for two reasons. First, formatting a log message typically takes 0.5-1 μs (1000-2000 cycles). In a low latency server, this could represent a significant fraction of the total service time for a request. Second, the I/O is expensive. Log messages are typically 50-100 bytes long, so a flash drive with 250 Mbytes/sec bandwidth can only absorb a few million messages per second. In addition, the application will occasionally have to make kernel calls to flush the log buffer, which will introduce additional delays.

The most common solution to these problems is to move the expensive operations to a separate thread. For example, I/O can be performed in a background thread: the main application thread writes log messages to a buffer in memory, and the background thread makes the kernel calls to write the buffer to a file. This allows I/O to happen in parallel with application execution. Some systems, such as TimeTrace in PerfUtils [49], also offload the formatting to the background thread by packaging all the arguments into an executable lambda and evaluating the formatting in the background thread.

Unfortunately, moving operations to a background thread only has a limited benefit, as the operations must still be carried out at runtime. If log messages are generated at a rate faster than the background thread can process them (either because of I/O or CPU limitations), then either the application must eventually block or it must discard log messages. Neither of these options is attractive. Discarding log messages can cause important information to be lost at inopportune times, and blocking can induce long tail latencies for applications. Blocking is particularly unappealing for low-latency systems as the long delay can give the appearance that the service has crashed.

In general, developers must ensure that an application doesn't generate log messages faster than they can be processed. One approach is to filter log messages according to their severity level; the threshold might be higher in a production environment than when testing. Another possible approach is to sample log messages at random, but this may cause key messages (such as those identifying a crash) to be lost. The final (but not uncommon) recourse is a social process whereby developers determine which log messages are most important and remove the less critical ones to improve performance. Unfortunately, all of these approaches compromise visibility to get around the limitations of the logging system.

And unfortunately, this is the limit of what most logging systems can provide. They can move the formatting/IO costs from the foreground to background and/or they can limit the amount of logging by dropping certain log messages. While these two mechanisms can improve the performance of the system over a more naïve implementation, they do not consider whether or not the formatting or IO costs need to be paid at runtime.

2.3 NanoLog's Observations

The design of NanoLog grew out of two observations about logging.

The first observation is that fully-formatted human-readable messages don't necessarily need to be produced inside the application. Instead, the application could log the raw components of each message and the human-readable messages could be generated later, if/when a human needs them. Many logs are never read by humans. In many cases, log messages are consumed primarily by other computers to perform analytics or aggregations. It's only on rare occasions that human developers consult the log to aid in debugging or auditing, and even then they only require small fraction of the log file to perform the task. So to optimize for the normal case, NanoLog proposes that a logging system should instead output the log data in a binary format, but also provide a mechanism to reconstitute the human-readable log messages at a later point.

The second observation is that log messages contain a lot of static log information that can be removed or deduplicated. For example, in the log message in Figure 1.1, the only dynamic parts of the message are the time, the thread identifier, and the values of the `name` and `tableId` variables. All other information is known at compile-time, unchanging, and repeated on every invocation of that logging statement. NanoLog proposes that the logging system should de-duplicate all static log information in the log file, and only output the dynamic log information in the normal case upon invocation. This both saves on resources (such as I/O and compute) at runtime, and it dovetails well with the previous proposal of a binary log file.

These two observations led to the overall design of the NanoLog system. NanoLog incorporates a compile-time component to extract the static log information and rewrite the log statements to output only dynamic information; it outputs a compressed, binary log file at runtime, and then utilizes a post-processor to recombine the static and dynamic information and form the full, human-readable log messages. Overall, the NanoLog system retains the familiar printf-style [6] logging API, but it adds a compile-time component to the user's build chain, and an optional post-processor to interpret the binary log files.

The remainder of this dissertation describes how NanoLog capitalizes on these observations to improve logging performance by 1-2 orders of magnitude.

Chapter 3

Overview

This chapter presents some of the key optimizations that enable NanoLog's nanosecond scale performance, and provides a brief overview of the resulting system architecture.

3.1 Key Techniques and Optimizations

The design of the NanoLog system is guided by four optimizations that enable its level of performance.

First, NanoLog avoids the cost of formatting log messages and defers it to a post-processor. It emits the raw components of a log statement (such as its format string and format arguments) in a binary format, and reconstructs the full human-readable representation at a later point with a separate application. This design was born from the observation that most log messages are never read by humans; thus the NanoLog system moves formatting the log messages out of the runtime hot path, freeing compute resources and reducing log latency.

Second, NanoLog separates the static and dynamic log information and places them on separate data paths. NanoLog makes the observation that portions of the log statement are known at compile-time and do not change throughput the execution of the application. For example, in Figure 1.1, the filename, line number, severity, and format string never change. This means that repeated invocations of the same log statement would result in multiple copies of this information in the log file. Thus, NanoLog extracts the static log information at compile-time, emits it just once at runtime, and rewrites the log statements to emit only dynamic log data. This technique amortizes the cost of the static log information and greatly reduces the I/O associated with repeated invocations of the same log message.

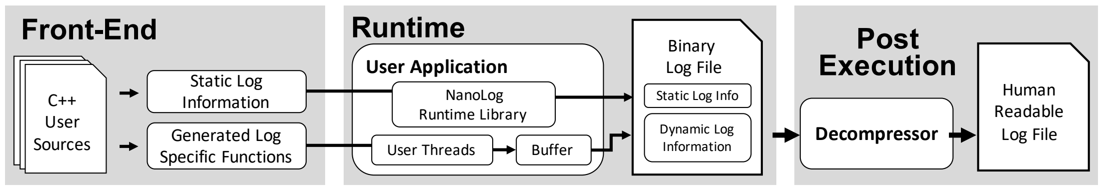


Figure 3.1: Overview of the NanoLog System. At compile-time, the NanoLog front-end extracts the static log information from the user sources and generates specialized logging functions that persist only the dynamic log information. At runtime, the NanoLog runtime outputs the static information just once into the binary log file, while the user threads execute the generated functions. These functions place the dynamic log data into a buffer and later the NanoLog library will output the buffer's contents to binary log file. At post-execution, the binary log file is passed into a decompressor application to generate a final, human-readable log file.

Third, NanoLog analyzes the log statements at compile-time and generates optimized logging functions. Without NanoLog, the printf API [6] would normally require the application to parse the format string and determine how to process the dynamic arguments at runtime. The operation requires the use of extremely branchy logic (i.e. code with a lot of if-statements), which does not synergize well with out-of-order execution engines and cache prediction algorithms in modern processors. NanoLog improves upon this by analyzing the format string at compile-time and generating highly optimized functions that are specialized to each log statement in the sources. This allows the runtime to execute in-line code with minimal branches, improving the overall performance of the system.

Lastly, NanoLog uses an extremely light weight compression scheme, called *variable length integer encoding*, to compress its log data. With the static log information removed, the bulk of NanoLog's remaining dynamic log data consists of integers. This allows the use of highly efficient, variable length encoding schemes as a form of compression. The scheme outperforms traditional, dictionary based algorithms both in terms of processing time and resulting output size.

3.2 Architecture

The NanoLog system consists of three components: a compile-time component, a runtime library, and a post-processor (Figure 3.1).

The compile-time component, *front-end*, optimizes the user application for nanosecond scale logging. It analyzes the user log statements at compile-time, extracts the static log information, and generates optimized functions for each log statement. The extracted static log information is organized into a *dictionary* (i.e. a lookup table) and given to the runtime library to persist just once at runtime. The generated functions are compiled into the application for the runtime to execute. The front-end generates two functions per log statement: a `record()` function that will persist the remaining dynamic log information into an in-memory buffer and a `compress()` function that will read the contents from the buffers and compress them to an output device. The `record()` function will replace the original log function in the sources, and `compress()` will be invoked by the runtime library. These functions allow the runtime to execute highly optimized, inline logic for each log statement.

The NanoLog *runtime* library buffers the dynamic log data and emits a binary log file at runtime. The runtime library is compiled into the application and executes within the same process. It provides low-latency *staging buffers* to store the application's dynamic log data, and a background thread to compress it for output. The log data is generated by user threads executing the `record()` function, and it is compressed via the `compress()` function. Additionally, the runtime manages the dictionary and ensures that the static information is written no more than once in each log file. The result of the execution is a *binary log file* that segregates the static and dynamic log data.

The *post-processor* or *decompressor* is a separate application that interprets the binary log file, recombines the dynamic log data with the static information in the dictionary, and presents it to the user for consumption. The post-processor can either be executed as a standalone application or compiled into another application as a library package. Executed as a stand-alone application, the post-processor will read the binary log file and output a human-readable log file. This mode is intended for human consumption of the log messages. Integrated as a library, the post-processor allows an external application (such as a log analytics application) to consume the binary log data without intermediate formatting steps. This allows for fast and efficient processing as the analytics platforms can process the data in binary without the need to inflate and parse ASCII log messages.

Overall, the NanoLog architecture differs from traditional logging systems in that it introduces a compile-time component, emits the log files in a binary format at runtime, and utilizes a post-processor to format the log files. This architecture shifts work out of the runtime hot-path and into the compile-time and post-execution phases of the application.

3.3 The Two Versions of NanoLog

There are two versions of NanoLog[1] in the GitHub repository [70]. The core philosophy of NanoLog is that it attempts to shift as much logging work as possible out of the application runtime. Some work can be performed once per log statement, either at compile-time or when the log statement is first invoked. Other work, we can defer until a post-processing stage. My first attempt at the NanoLog system took this idea to the extreme. It utilized a separate preprocessor to read and modify the user sources at compile-time, and maximally extract work and I/O. However, as the project progressed, the C++17 standard was being finalized with fairly strong compile-time computation support. I realized that one could get most of the benefits of preprocessor NanoLog by using C++17 metaprograming features instead of a preprocessor. Thus, two versions of the NanoLog system were born: the *preprocessor* version of NanoLog that provides the maximal performance and the *C++17* version of NanoLog that provides highest ease of use. Both versions provide the core functionality extracting static log information and generating highly optimized functions, but their front-end mechanisms differ. Both versions will be described in detail starting in Chapter 4.

3.4 Dissertation Outline

The rest of this dissertation will describe the NanoLog system in detail. Chapters 4, 5, 6 will describe the internal architectures of the front-end, runtime, and post-processor respectively. Chapter 7: *The Staging Buffers: A Cache Conscious Design* further breaks down the challenges of minimizing cache misses at runtime to enable nanosecond scale logging, and Chapter 8: *Compression* examines the compression scheme used in NanoLog. Finally, Chapter 9: *Evaluation* evaluates the performance of NanoLog with a proof-of-concept implementation on GitHub [70], Chapter 10 discusses the limitations of and extensions one could build for NanoLog, and Chapter 11 discusses other work in the field related to NanoLog. Finally, the dissertation concludes with Chapter 12.

[1]The overview architecture described the common functionality between the two.

Chapter 4

Front End

The goal of the NanoLog front-end is to reduce the cost of executing log statements at runtime by performing optimizations at compile-time. In particular, it reduces the amount of I/O required for logging by building a dictionary to de-duplicate static log information, and it reduces the run-time compute required to process each log statement's remaining dynamic arguments by generating specialized code to process the arguments for each log statement. There are two versions of the NanoLog front-end: Preprocessor NanoLog and C++17 NanoLog. Both will be discussed in full detail in the subsequent sections; this section covers the common functionality between the two.

The NanoLog system reduces I/O required for logging by placing the static and dynamic log information on separate data paths. The static, non-changing information is placed into a *dictionary* and persisted just once in the log file, while the dynamic information is persisted per log invocation with a reference back to the dictionary. Figure 4.1 shows an example of how this is done. The log messages are shown in their human-readable format in the middle of the figure, and each log statement contains a substantial amount of static, non-changing information. This information includes the filename/line number tuple identifying where a log message occurred, the severity level of the log message, and most of the original format string (i.e. portions that do not contain % specifiers). Traditional logging systems are forced to repeatedly output this redundant information when the log statement is invoked multiple times. This is done to make the logs immediately available in a human-readable format. NanoLog instead works to output a binary log file similar to the bottom of the figure; the static information is de-duplicated and represented just once in the dictionary, and all future messages include only the dynamic information with a reference to the dictionary information. This makes the log file smaller and saves I/O at runtime.

The NanoLog dictionary maps a 4-byte integer identifier to the static log information (filename,

Source Code

```
for (int id : tables)
        NANO_LOG(NOTICE, "Creating table ’%s’ with id %d", names[id], id);
```

Human Readable Log File

```
TableManager.cc:1031 NOTICE: Creating table ’orders’ with id 11
TableManager.cc:1031 NOTICE: Creating table ’acquisitions’ with id 23
TableManager.cc:1031 NOTICE: Creating table ’clients’ with id 1
TableManager.cc:1031 NOTICE: Creating table ’inventory’ with id 12
TableManager.cc:1031 NOTICE: Creating table ’upcs’ with id 14
```

Binary Log File (with Dictionary)

```
{1 => {TableManager.cc, 1031, NOTICE, "Creating table ’%s’ with id %d"}}

{1, "orders", 11}
{1, "acquisitions", 23}
{1, "clients", 1}
{1, "inventory", 12}
{1, "upcs", 14}
```

Figure 4.1: The top portion shows source code for an application repeatedly invoking a log statement in a loop. The middle portion shows a possible representation of the human-readable log file produced by a traditional logging system. Notice that it contains repetitive, non-changing static information in the form of “`TableManager.cc:1031 NOTICE: Creating table ...`” The bottom portion shows a different representation (similar to NanoLog) that uses a dictionary to de-duplicate the static log information. The first line in bold is the dictionary; it maps the integer “1” to the log message’s static information. The subsequent lines represent the dynamic arguments for each log invocation with a reference back to the dictionary (“1”).

line number, severity, and format string). The mapping is dense in that each identifier, called the *unique log identifier*, only maps to exactly one log message in the original source code and the range of identifiers is contiguous starting at zero. The dense packing enables further compression, which is discussed in Chapter 8. The dictionary entries are built at compile-time by preprocessing the log statements and are emitted in the log file before their first use by the runtime log statements. This ordering ensures that the static information is available before use and allows the statements to simply emit a 4-byte integer to refer to the static information. Using this scheme, NanoLog reduces the I/O required to persist repeating log statements by collapsing the static information into a 4-byte integer reference.

The NanoLog system also generates specialized functions at compile-time to reduce the runtime compute required to process the statement's arguments. The printf logging API, which NANO_LOG is modeled after, allows the application developer to include a variable number of arguments with varying data types [6]. Traditionally, to process the arguments, logging systems will parse the format strings at runtime to infer the types of the arguments. However, parsing the format strings is an expensive operation; it takes on the order of microseconds to complete. NanoLog eliminates this cost by preprocessing the log statements at compile-time and generating functions specialized to each log statement.

The NanoLog front-end generates two functions for each NANO_LOG statement. The first function, `record()`, is invoked in place of the original `NANO_LOG()` statement. It copies all the dynamic information (i.e. the function parameters) passed into the initial NANO_LOG () invocation to an in-memory buffer[1]. The second function, `compress()`, is invoked by the NanoLog background compaction thread to compress the recorded data for more efficient I/O. The two functions are intrinsically linked to a specific log statement and to each other; `compress()` consumes the data produced by `record()` and `record()` is specialized to a specific log statement's argument types.

Both functions prepare a log statement's dynamic data for output, but they are separated to reduce application latency. More specifically, the `record()` function is invoked directly by the application logging thread so it's optimized to perform as little compute as possible. The `compress()` function contains heavier weight computation to compress the arguments and is invoked by the NanoLog background thread. The intermediate buffering allows both operations to operate in parallel on separate threads, and the buffering hides the compression latency for bursts of logging activity.

[1]The "memory buffer" is the Staging Buffer as discussed in Chapter 7.

```
inline void record(buffer, id, name, tableId) {
  buffer.push<int>(id);
  buffer.pushTime(RDTSC());
  buffer.pushString(name);
  buffer.push<int>(tableId);
}

inline void compress(char **buffer, char **out) {
  pack<int>(buffer, out);  // logId
  packTime(buffer, out);   // time
  packString(buffer, out); // string name
  pack<int>(buffer, out);  // tableId
}
```

Figure 4.2: Conceptual psuedo-code that is generated by the NanoLog front end for the log message in Figure 4.1. The `record()` function stores the dynamic log data to a buffer and `compress()` compacts the buffer's contents to an output character array.

Compression is discussed further in Chapter 8 and the performance impact of buffering is analyzed in Chapter 9.

Figure 4.2 shows slightly simplified versions of the functions generated for the NANO_LOG statement in Figure 4.1. The `record()` function performs the absolute minimum amount of work needed to save the log statement's dynamic data in a buffer. The invocation time is read using Intel's RDTSC instruction, which utilizes the CPU's fine grain Time Stamp Counter [46]. The only static information it records is a unique log identifier for the NANO_LOG statement, which is used by the NanoLog runtime background thread to invoke the appropriate `compress()` function and by the decompressor to retrieve the statement's static information. The arguments of the recorded function match the original NANO_LOG statement. The types of `name` and `tableId` were determined at compile-time by the preprocessor by analyzing the "`%s`" and "`%d`" specifiers in the format string, so `record()` can invoke type-specific methods to record them.

The purpose of the `compress()` function is to reduce the number of bytes occupied by the logged data, in order to save I/O bandwidth. The preprocessor has already determined the type of each item of data, so `compress()` simply invokes a type-specific compaction method for each value. Chapter 8 discusses the kinds of compaction that NanoLog performs and the trade-off between compute time and compaction efficiency.

4.1 Implementations

The NanoLog front-end for C++ has been implemented in two ways. Preprocessor NanoLog uses a Python script to scan through the use sources at compile time and inject optimized functions directly into the user sources that the runtime can blindly invoke. C++17 NanoLog utilizes C++17's strong meta-programming support to generate the optimized functions as inline templated code. Each has its own strengths/weaknesses and unique set of challenges to overcome. The rest of this section will focus on the implementation differences, challenges, and impact on performance.

4.2 Preprocessor NanoLog

The preprocessor version of Nanolog represents the most performant version of NanoLog; it takes moving computation and data outside of the runtime to the extreme. It analyzes the log statements at compile-time, generates the complete chain of logic that encodes how to and in which order to process the dynamic log arguments for each log statement, and then injects them into the user application, NanoLog runtime, and post-processor. By encoding this knowledge directly into the logic, the preprocessor version eliminates the need to parse or output the static information during the runtime portion of the application, saving I/O and compute.

The rest of this section describes Preprocessor NanoLog's architecture, the structure of the files it generates, and the challenges associated with its design.

4.2.1 Preprocessor NanoLog Architecture

The preprocessor version of NanoLog interposes on the compilation process of the user application (Figure 4.3). It processes the user source files with a Python script and generates a *metadata file* and a modified source file for each. The modified source files are then compiled into object files. Before the final link step for the application, the NanoLog combiner reads all the metadata files and generates an additional C++ source file that is compiled into the NanoLog runtime library (Chapter 5) and the post-processor (Chapter 6). This library is then linked into the modified user application to form the final executable.

To ensure seamless operation for the user, the NanoLog system interposes on the compilation process by providing the user with a build macro that serves as a drop-in replacement for invocations to the compiler. This macro will invoke a Python script (known as the *preprocessor*) on the source file, compile the modified sources, and delete the modified sources before returning. The

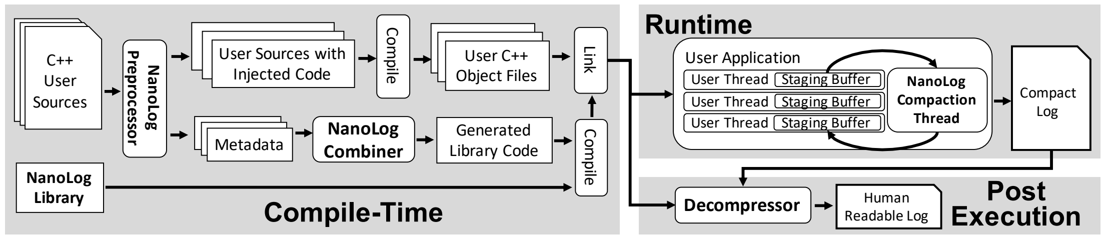


Figure 4.3: Overview of the NanoLog Preprocessor Front-End. At compile time, the user sources are passed through the NanoLog preprocessor, which injects optimized logging code into the application and generates a metadata file for each source file. The modified user code is then compiled to produce C++ object files. The metadata files are aggregated by the NanoLog combiner to build a portion of the NanoLog library. The NanoLog library is then compiled and linked with the user object files to create an application executable and a decompressor application. At runtime, the user application threads interact with the NanoLog staging buffers and background compaction thread to produce a compact log. At post execution, the compact log is passed into the decompressor to generate a final, human-readable log file.

latter deletion is to hide the fact that the user sources are modified, so the user only sees their original, unmodified sources. Section 4.3.1 explains in more detail how this is done in the GNUmake environment [59] and Section 4.3.6 describes how the system manipulates the compiler to report errors in the original file location in the face of source modifications.

The preprocessor script separates the logic to process a single log statement into three functions and optimizes the logic so that the static information is never needed at runtime. For every log statement encountered by the preprocessor, it generates three highly coupled functions that are specialized to that log statement. There is the `record()` function that stores the log statement's dynamic arguments into an in-memory buffer, the `compress()` function that reads the arguments back and compresses them from output, and the `inflate()` function that reads the compressed arguments and formats them for human-readable output (Figure 4.4). These functions encode the order and types of the dynamic arguments directly in their logic, and the inflate() function inlines the static data needed to output the final human-readable string directly in its logic. With this design, the three functions never have to persist or refer to the static arguments like the format string to perform their operations, saving both I/O and compute. The record() function is injected into the user sources (Section 4.3.4) for execution in lieu of the original log statement, while the compress and inflate functions are placed in a metadata file (one for each source file) for later processing.

The metadata files are used to track static log information across separately compiled source files. The NanoLog system needs to collate all the static log information to generate the final C++

Preprocessor Code Generated

```
inline void record(buffer, name, tableId) {
  ...
  buffer.push<int>(_logId_TableManager_cc_1031_Creating_table_ps_with_id_pd);
  buffer.pushTime(RDTSC());
  buffer.pushString(name);
  buffer.push<int>(tableId);
}

inline void compact(char *in, char *out) {
  pack<int>(in, out);   // logId
  packTime(in, out);    // time
  packString(in, out);  // string name
  pack<int>(in, out);   // tableId
}

inline void inflate(char *in, void (*aggregate)(...)) {
  // Static info directly embedded in source
  const char *fileName  = "TableManager.cc"
  const int lineNum     = 1031;
  const char *logLevel  = formatLogLevel(NOTICE);
  const char *fmtString = "Creating table '%s' with id %d";

  // Dynamic information extraction
  int id                = unpack<int>(in);
  long time             = unpackTime(in);
  const char *arg1      = unpackString(in);
  int arg2              = unpack<int>(in);

  ...

  // Context + Log Message
  printf("%s %s:%d %s: ", time, fileName, lineNum, logLevel);
  printf(fmtString, arg1, arg2);
}
```

Figure 4.4: Sample code generated by Preprocessor NanoLog for the log statement "`Creating table '%s' with id %d`" in file "TableManager.cc" on line 1031. The `record()` function stores a log statement's dynamic arguments to an in-memory buffer, the `compress()` function reads the arguments back and compresses them to an I/O buffer, and the `inflate()` function reads the compressed arguments and formats them. Each encodes the type and order of the arguments in its logic, and the `inflate()` function additionally includes the static log information (filename, line number, and format string) in its source code.

file for the runtime library and post-processor. A strawman approach to this would be to use a single execution of the preprocessor script to parse through all the sources, collect the static log information, and generate the final C++ file at every compilation of the user application. However, this method would be extremely slow as it would utilize only a single thread and reprocess all the log statements with every compilation. In practice, the user sources are often compiled in parallel, and most compilations typically only require the modified files to be recompiled. Thus, the NanoLog uses the paradigm of emitting a single metadata file per source file and then collating them to generate the final library file. This architecture allows parallel instances of the Preprocessor to execute without contention on a global data structure, and the system to reuse the metadata files for sources that do not need to be recompiled.

The NanoLog combiner links together the generated record/compress/inflate functions across the user application, runtime, and post-processor. It does this by collating all the metadata files and creating a global ordering of all the log statements. It then uses this canonical ordering to assign a unique 4-byte integer to each log message and generate compress() and inflate() function arrays to match. The integer is communicated to the record() function via an assignment to an C extern variable in the generated library file (`_logId_TableManager_cc_1031_Creating_table...` in Figure 4.4; Section 4.3.5 discusses this in more detail) and is persisted as part of the dynamic log data for every log invocation. The runtime and post-processor can then use this integer as an index into the compress() and inflate() function arrays to ensure that the correct processing logic is invoked. With this strategy, the application logic will always know which function to invoke based on the integer and will never have to process the static data associated with each log message.

The final output of the NanoLog combiner is a single C++ file that contains the mapping of extern variables to integer assignments, the generated compress()/inflate() functions, and two function arrays containing the compress()/inflate() functions (Figure 4.9). This C++ file is then compiled into the user application/NanoLog runtime and post-processor, so that the record() functions can refer to the extern assignments and the runtime/post-processor can use the assignment to index the correct compress()/inflate() function for each log message.

One caveat of this design is that it ties a specific compilation of the user application to a specific compilation of the NanoLog runtime library and post-processor. The reason is that the three functions used by each component (record() for the user application, compress() for the library, and inflate() for the post-processor) are tightly coupled, and linked together via an assignment of a 4-byte integer identifier by the NanoLog combiner. Different compilations of even the same application can cause the assignment to shift, potentially causing a mismatch between the integer persisted by the record() function in the application and the ordering of the compress()/inflate() function arrays in the runtime/post-processor. As a result, every recompilation of the user application, no matter how minor, must trigger a recompilation of the runtime library and post-processor, and an exact version of the post-processor must be used with an exact version of the application to interpret the log files. This can potentially cause operational nightmares as users must persist both log files and *specific versions* of the post-processor application to interpret the logs; any mismatch can cause the logs to be indecipherable.

To address this usability concern, the preprocessor can decouple the post-processor from the application by generating an additional runtime function, `writeDictionary()`, to dump the mapping of integer assignments to static log data in the log file. This function is used by the runtime library to output this mapping just once at the beginning of the log file during its execution, and the post-processor can choose to discard its internal mapping of the inflate functions in lieu of using the version encoded in the log file instead. This design does incur a slight performance cost as the runtime must output this mapping/dictionary at runtime, and the post-processor no longer uses its highly optimized `inflate()` function. However, I believe this overhead is worth the increased usability, so this behavior is enabled by default. Furthermore, the preprocessor version of NanoLog still retains its highly optimized `record()`/`compress()` function chain, the cost of `writeDictionary()` is amortized, and the post-processor's performance is less important in enabling nanosecond scale logging at runtime.

4.3 Preprocessor Challenges and Implementation Details

This section of the dissertation details some of the more mundane challenges and operations of the preprocessor system. It additionally shows the layout of some of the functions/files generated by the NanoLog preprocessor and combiner. Of particular interest may be Figure 4.7 which shows the modifications performed to the user sources, Figure 4.8 which shows the JSON metadata file generated per user source file, and Figure 4.9 which shows the final C++ source file generated by the NanoLog combiner.

4.3.1 Interposing on Compilation

To properly interpose on the compilation process, the NanoLog system imposes three requirements for the users' build environment. First, the build system must be able to invoke the NanoLog

GNUmake Macro

```
 1 # Configuration parameters to store NanoLog's include and preprocessor directories.
 2 INCLUDE_DIR=$(NANOLOG_DIR)/runtime
 3 PREPROC_DIR=$(NANOLOG_DIR)/preprocessor
 4
 5 # run-cxx:
 6 # Compile a user C++ source file to an object file using the NanoLog system.
 7 # The first parameter $(1) should be the output filename (*.o)
 8 # The second parameter $(2) should be the input filename (*.cc)
 9 # The optional third parameter $(3) should be additional options compiler options.
10 # The optional fourth parameter ($4) should be gnu preprocessor options.
11 define run-cxx
12         $(CXX) -E -I $(INCLUDE_DIR) $(2) -o $(2).i -std=c++11 $(4)
13         python $(PREPROC_DIR)/parser.py --mapOutput="generated/$(2).map" $(2).i -o $(2).ii
14         $(CXX) -I $(INCLUDE_DIR) -c -o $(1) $(2).ii $(3)
15         @rm -f $(2).i $(2).ii
16 endef
17
18
19 # GNUmake rule demonstrating run-cxx's usage for main.cc
20 main.o: main.cc
21         $(call run-cxx, main.o, main.cc, -DNDEBUG -O3 -g)
```

Figure 4.5: The GNUmake macro that serves as a replacement for the user's compiler when compiling with Preprocessor NanoLog. The macro processes exactly one user source file (*.cc) and outputs a C++ object file (*.o). Starting on line 12, the macro will run the GNU preprocessor on the user source file, `$(2)`, and save the output to `$(2).i`. It then invokes the NanoLog preprocessor on `$(2).i` and saves the results to `$(2).ii`. Finally, it invokes the GNU compiler on the NanoLog generated source to generate the output file, `$(1)`, and cleans up the generated files, `$(2).i` and `$(2).ii`. In this figure, the "`$(CXX)`" invokes the GNU C++ compiler, and "`generated/$(2).map`" stores the metadata associated with this source file.

preprocessor script on all the source files and compile the modified sources in lieu of the original sources. Second, the system must be able to detect when all the sources have been preprocessed and invoke the NanoLog combiner to generate the `record()`, `compress()`, `inflate()`, and `writeDictionary()` functions for the NanoLog library. Finally, the library must be recompiled with the new functions and linked into the user application.

The NanoLog implementation on GitHub[70] demonstrates how these three requirements can be satisfied using the GNUmake build system[59] and compiler. In this implementation, it is assumed that the user will compile each source file individually into an object file before linking them into the final application, and that all source (and thus object) files can be identified in the script a priori.

To ensure that all sources pass through the NanoLog preprocessor, the NanoLog system provides a GNUmake macro (Figure 4.5) that serves as a drop-in replacement for invocations to the GNU g++ compiler. This macro will invoke the preprocessor with the original sources passed in, invoke the compiler on the modified sources, and finally clean up by removing the modified sources. By using this macro instead of invoking the compiler directly, the user can ensure that all sources will pass through the NanoLog preprocessor.

To satisfy the second and third requirements of generating and rebuilding the NanoLog utility functions and library respectively upon source changes, the system builds the following dependency chain in the GNU makefile: (executable) →(NanoLog Library) →(Generated Functions File) →(User Object Files) →(User Sources) where "→" indicates "depends on". This dependency chain ensures that all object files are compiled and up-to-date before generating the utility functions and linking the NanoLog library into the user application. Additionally, this chain ensures that any changes to the base source files will lead to the same cascade of operations.

4.3.2 Identifying the Log Statements

To process the log statements in the user sources, the NanoLog preprocessor must first correctly identify NANO_LOG statements in the code and exclude false-positives. Unlike other C++ functions, NANO_LOG is not a real function; it is only an ASCII text marker used by the user to tag log-like functions to be replaced by the NanoLog preprocesor. Thus, the preprocessor does not try to semantically understand the code to find the NANO_LOG statements. Instead, it simply searches through the sources for the "NANO_LOG" string. This method, however, can generate false-positives such as the NANO_LOG string appearing in comments, C-style macros, or as a part of a longer C/C++ identifier such as `bool ENABLE_NANO_LOG`.

```
/* This is just a comment that contains NANO_LOG() in it. */
NAN\
O_LOG(severity, "It has commas, line br\
eaks "

"and string concatinenatinons %d, %d %s"
, functionCall(1, func(2, 3), 4), [a, b] {
 return a + b; }
" Another B\
reak and \"embedded\" \"'s");
```

```
NANO_LOG(severity,
        "It has commas, line breaks, and string concatinenatinons %d, %d %s",
        functionCall(1, func(2, 3), 4),
        [a, b] { return a + b; },
        " Another Break and \"embedded\" \"'s");
```

Figure 4.6: This example represents a convoluted but valid NANO_LOG statement (top) that the preprocessor must properly parse and interpret. The statement contains line breaks in the function invocations, lambda invocations, and strings. Additionally, it contains a NANO_LOG statement in the comments that must be sanitized. The bottom portion shows a sanitized version of the log statement for the reader's sanity.

The NanoLog preprocessor sanitizes the user sources by first passing them into the GNU C-preprocessor. This trivially eliminates all comments from the sources and resolves odd syntax such as line breaks in the middle of identifiers. In addition, all macros will be expanded and inlined, which eliminates false matches to the NANO_LOG substring appearing in macro names, and additional instances of NANO_LOG invocations embedded in the macros will be exposed for the preprocessor to detect. This latter step is important since every instantiation of a macro can cause new NANO_LOG statements with a different filename/line number to be injected. Passing the sources through the GNU preprocesor effectively sanitizes the sources and ensures that the NanoLog preprocessor only has to work with well-formed, inlined code.

With the sources sanitized, the NanoLog preprocessor must search for valid NANO_LOG invocations and separate its arguments. To detect whether a particular NANO_LOG substring is a valid invocation, it uses a set of heuristics to ensure it matches the structure of a invocation. In particular, it ensures that the NANO_LOG substring is not a part of a longer identifier (i.e. such as `bool ENABLE_NANO_LOG`) by requiring the first character preceding the substring to be non-alphanumeric and requiring the next non-whitespace character following the substring to be a left parenthesis "`(`". This ensures that the substring matches the structure of a function invocation.

To separate the NANO_LOG invocation's argument list, the preprocessor uses a heuristic of delimiting by top-level commas. Although most argument lists can be delimited by commas alone, the top-level distinction is important since commas can be embedded in string literals and lambda/-function invocations. Figure 4.6 shows an exceptionally difficult argument list to parse. To ensure that commas are top-level, it only considers commas that are not contained with parentheses, curly braces, brackets, or quotations for string literals. It does this by keeping a depth count for each (incremented when an open is detected and decremented when a close is detected) and only using a comma as a delimiter if the depth counters are zero. Special care is taken to detect if these special characters are detected within a string literal, and those instances will not count towards the depth counters. This heuristic ensures that the commas are top-level, and allows the preprocessor to separate the argument list.

After the statements are identified and the arguments separated, the preprocessor will next verify that the arguments are well formed before generating additional source code. In particular, the NanoLog system will attempt to parse the invocation's arguments and ensure that (a) it contains a literal format string and (b) the number of arguments following the format string matches the number of specifiers required by the format string. If either of these cases is false, an error is raised (more about this in Section 4.3.6).

After all the NANO_LOG invocations have been parsed, the preprocessor can now start to generate the replacement functions.

4.3.3 Generating the Functions

For every log statement encountered, the NanoLog preprocessor generates four tightly coupled functions and a C-style identifier that ties them together. The C-style identifier, called a *tag*, is a string formed from combining the log statement's filename, line number, and format string, and uniquely identifies a single log statement in the sources. For example, the message "`Hello World %d`" in main.cc on line 4 will have the tag "`uniqueId_main_cc_4_Hello_World_pd`". This tag is used in the dictionary to map unique instances of log messages to the generated functions and to detect when a log message has already been encountered by the preprocessor [2].

The four functions generated per log message are `record()`, `compress()`, `inflate()`, and `writeDictionary()`. These functions are unique to exactly one log statement in the source files and are used to process that specific log statement's arguments. The record and compress functions are used by the runtime to persist a log statement's arguments and compress them for output to disk

[2]Log messages can be duplicated in the sources via #include-ing header files. This is described in more detail in Section 4.3.5.

main.cc

```
 1  #include "NanoLog.h"
 2
 3  main() {
 4    NANO_LOG(DEBUG, "Hello World %d", rand());
 5  }
```

Modified main.cc

```
 1  // NanoLog.h is inlined here
 2  # 1 "NanoLog.h"
 3
 4  ...
 5
 6  // Marks the end of NanoLog.h (inlined as part of the header)
 7  static const int __internal_dummy_variable_marker_for_code_injection = 0;
 8
 9  // Start of injected functions
10  # 1 "generated.h"
11  inline void
12  record_main_cc_4_Hello_World_pd(int severity, const char *str, int arg1) {
13    extern int uniqueId_main_cc_4_Hello_World_pd;
14
15    NanoLog::push(uniqueId_main_cc_4_Hello_World_pd);
16    NanoLog::push(RDTSCP());
17    NanoLog::push(arg1);
18  }
19
20  // Start of the user logic
21  # 3 "main.cc"
22  main() {
23    record_main_cc_4_Hello_World_pd(
24  # 4 "main.cc"
25            DEBUG, "Hello World %d", rand());
26  }
```

Figure 4.7: A sample application before (top) and after (bottom) it is processed by the NanoLog preprocessor. The top figure contains a simple application that logs only an integer. The bottom portion shows how the NanoLog header is inlined by the GNU preprocessor in lines 1-8, where the `record()`definition is in injected on lines 9-19, and where the user application is modified to invoke the `record()`function in lieu of NANO_LOG on line 26. The lines starting with "#" are GNU preprocessor directives that inform the GNU compiler where within a source file the next line belongs. For example, "`# 4 "main.cc"`" (line 25) informs the compiler that the next line (line 26) originated from line 4 of main.cc.

as described in the introduction of this chapter. The writeDictionary and inflate functions are used to communicate the static information to the post-processor in different ways; the inflate function encodes the static information for the log message in code form, while the writeDictionary function emits it in a data form. The record function is injected into the sources (Figure 4.7), while the remaining three are placed into a metadata file for latter consumption by the NanoLog combiner (Figure 4.8). Section 4.3.4 discusses how the injection is performed, while Section 4.3.5 discusses the structure of the metadata file.

One odd construct I'd like to draw attention to is the extern integer variable in the record() function. This extern integer is declared in line 13 of Figure 4.7 and immediately used in line 15. This integer serves as an application-wide unique log identifier for the particular log statement, and connects the four generated functions. It is ultimately used by the runtime and (optionally) post-processor to index into the compress()/inflate() function arrays generated by the NanoLog combiner. Here, it is declared as an extern integer so that the value of the integer can be assigned much later by the NanoLog combiner after it has de-duplicated log statements encountered multiple times by the preprocessor. The value of the extern integer is ultimately defined in the generated library file (Figure 4.9).

writeDictionary() vs. inflate()

The writeDictionary() function communicates the static information via data, while the inflate() function communicates the static information via code. More specifically, the writeDictionary() function can be used by the NanoLog runtime to output a dictionary of static information at the start of the log file. The post-processor can then parse this dictionary and build the appropriate internal structures to interpret and decode the remaining log file (further discussed in Chapter 6).

The inflate() function, on the other hand, communicates static information via code. In particular, each inflate() function encodes all the necessary information to decode a specific log statement in its logic, and this logic is directly compiled into the post-processor. As seen in Figure 4.4, the inflate() function contains the static ASCII log information and the operations needed to extract the dynamic arguments from the log file. Compiling the inflate() functions into the post-processor results in a highly performant system, as it contains no branching and allows the compiler to aggressively optimize the operations.

However, while the inflate() method is more performant, the NanoLog system preferentially uses the writeDictionary() method as it is more user-friendly. In particular, the writeDictionary() method encodes all the information needed to decode the log file in the log file itself. This means that a single NanoLog post-processor application can decode all NanoLog log files. The inflate() method, on the other hand, ties a *specific compilation* of an application, with a *specific compilation* of the post-processor. This is the case since a recompilation of the application can cause the 4-byte unique log identifiers to shift and thus no longer map to the correct inflate() method in the post-processor. Operationally, this means that in addition to the log file, the user must also persist a specific version of the post-processor to be used with the log file. This can be an operations nightmare for the users of NanoLog, so NanoLog sacrifices a bit of performance at post-execution for usability and flexibility by using the writeDictionary() method by default.

4.3.4 Injecting the record() Function

The `record()`function is intended to be injected into the user sources, and invoked in place of the original NANO_LOG statement. This requires the function to be defined before first use and have a signature compatible with the arguments passed into the original NANO_LOG statement. To ensure that the functions are defined before first use, the NanoLog preprocessor injects the `record()`functions at the beginning of the user source files. To ensure that the function parameters are compatible, NanoLog generates the `record()`function using the types encoded in the format string.

The NanoLog preprocessor injects the `record()`functions immediately after the `#include "NanoLog.h"` line in the user's sources. Since it is C++ convention to include a system's header before using any of the library functions, the NanoLog system treats NANO_LOG in a similar way. In particular, the preprocessor will search the source file for the logical line immediately after the `#include "NanoLog.h"` statement and inject all the record functions generated for this source file there. This ensures that the functions are defined before first use/first invocation to NANO_LOG and is consistent with the expectations of traditional C++ libraries.

In practice, it can be difficult for the preprocessor find the line after `#include "NanoLog.h"`, so it uses a special marker to help it. The current implementation of NanoLog[70] requires the sources to first pass through the GNU C-preprocessor to sanitize comments before passing through the NanoLog preprocessor. This GNU preprocessing step causes the entire header file to be inlined into the sources, making it difficult to identify exactly where to inject the record function definitions. To aid with the problem, the NanoLog header includes a special variable with a long and obscure identifier at the end of the file, such as `__internal_dummy_variable_marker_for_code_injection` in Figure 4.7. This variable effectively marks the end of the header file. The preprocessor can then scan for this special variable and inject the record definitions there. This essentially allows for behavior equivalent to injecting the record function definitions after the `#include "NanoLog.h"` statement.

The NanoLog preprocessor analyzes the order and types of the format specifiers in the original format string to generate a `record()` function with a signature that matches the original NANO_LOG statement. For example, a `NANO_LOG(DEBUG, "Hello \%s, you are user \%d", ...)` would generate a record function with the signature `record(severity, const char*, const char*, int)`. The first two arguments are always required by the NANO_LOG API to specify the severity level and format string, and the next two are generated based on the `%s` and `%d` specifiers present in the format string. A full mapping of format specifiers to argument types can be found in the printf documentation [6].

After these two constraints are satisfied, the original NANO_LOG statement can be replaced with an invocation to the generated function instead as seen in Figure 4.7.

4.3.5 Metadata files and the NanoLog Combiner

The NanoLog system uses consistent variable naming, metadata files, and a metadata file combiner to detect duplicated log statements and produce the generated library file.

User log messages can potentially be duplicated by the C++ compiler. In the C++ language, when a header file is `#include`-d in a source file, the compiler internally copies the entire contents of the header file into the source file before compiling. This means that log statements that exist in the header file will be effectively copied verbatim into every source file that includes it. This is problematic for NanoLog as it increases the amount of redundant generated code, takes up unique log identifier space, and creates naming conflicts.

NanoLog solves this problem by utilizing metadata files to track log statements across source files and consistent variable naming to detect duplication. As the NanoLog preprocessor processes the user source files, it generates one unique metadata file per source file (headers files not included) and places them in a special directory. Contained within each metadata file is a JSON object mapping unique log message *tags* to the four generated functions and the static information. The tag is based on a deterministic mangling of the log statement's filename, line number, and format string, and thus uniquely identifies a single log statement in the user sources. The tag is appended to every variable and function name generated by the preprocessor. This means that duplicated log statements in the sources will have the same variable and function names. This allows the NanoLog combiner to detect duplicate variables/functions based on naming alone. Figure 4.8 shows an example of the metadata file generated for the sources in Figure 4.7, and "`main_cc_4_Hello_world_pd`" is the unique log identifier tag appended to every generated function and variable name.

Metadata File for main.cc

```
{
  "count":1,
  "logId2Code": {
    "main_cc_4_Hello_World_pd":
      {
        "compressFnName":"compress_main_cc_4_Hello_World_pd"
        "inflateFnName":"inflate_main_cc_4_Hello_World_pd"",

        "compressFnDef":"size_t compress_main_cc...",
        "inflateFnDef":"..."

        "dictionaryFragment":"pushString(buffer, \"Hello World %d\");
                              pushString(buffer, \"main.cc\");
                              pushInt(buffer, 4);
                              pushInt(buffer, DEBUG);",

        "compilationUnit": "main.cc",
        "fmtString":"Hello World %d",
        "linenum":4,
        "logLevel":"DEBUG"
      },
  }
}
```

Figure 4.8: A simplified version of the metadata file generated by the NanoLog preprocessor for the source file in Figure 4.7. The metadata contains one large JSON object that contains a mapping (logId2Code) of unique log identifier tags for log messages to their associated generated code. "`main_cc_4_Hello_World_pd`" is the tag for the log message in Figure 4.7. Within the object, the `*FnName` variables store the function names for the compress/inflate functions, the `*FnDef`'s store the functions' definitions, and the `dictionaryFragment` stores the C++ code fragment to be used in the mega-writeDictionary() function. Lastly, the ellipses in `compressFnDef` and `inflateFnDef` correspond to the record/compress function definitions in Figure 4.9.

GeneratedCode.cc

```
 1  // Compress function definition for main.cc:4 - "Hello World %d"
 2  inline void compress__main_cc_4_Hello_World_pd(const char **in, char **out)
 3  {
 4      packTime(in, out);       // compress time
 5      pack<uint32_t>(in, out); // compress logid
 6      pack<int32_t>(in, out);  // compress %d argument
 7  }
 8
 9  // Inflate function definition for main.cc:4 - "Hello World %d"
10  inline void inflate_main_cc_4_Hello_World_pd(const char **in, FILE *outputFd,
11                                              void(*aggFn)(const char*, ...)) {
12    const char *filename = "main.cc";
13    const int linenum = 4;
14    const NanoLog::LogLevel logLevel = NanoLog::DEBUG;
15    const char *fmtString = "Hello World %d";
16    ...
17    int arg1 = readInt(in);
18    ...
19    if (outputFd)
20        fprintf(outputFd, fmtString, arg1);
21    if (aggFn)
22        (*aggFn)(fmtString, arg1);
23  }
24
25  // Assignment of 4-byte integers to unique log identifiers
26  extern const int uniqueId_main_cc_4_Hello_World_p = 0;
27
28  // Compress() and inflate() arrays reflecting the assignment
29  void (*compressFnArray[1]) (const char **in, char** out)
30  {
31    compress__main_cc_4_Hello_World_pd,
32  }
33  void (*inflateFnArray[1]) (const char **in, FILE *outputFd,
34                             void (*aggFn)(const char*, ...))
35  {
36    inflate_main_cc_4_Hello_World_pd,
37  }
38
39  // writeDictionary function that emits the static log information
40  long int writeDictionary(char **buffer, char *endOfBuffer)
41  {
42    // Code generated for "Hello World %d" in main.cc:4
43      pushString(buffer, "Hello World %d");   // Format String
44      pushString(buffer, "main.cc");          // filename
45      pushInt(buffer, 4);                     // Line number
46      pushInt(buffer, DEBUG);                 // Severity
47    ...
48  }
```

Figure 4.9: A simplified snippet of the library file generated for the source file in Figure 4.7 by the NanoLog preprocessor and combiner. The first two functions are the `compress()`and `inflate()`functions for the log statement, line 25 shows the `extern` integer assignment, lines 27-35 shows the function arrays for `compress()`/`inflate()`, and the last function outputs the dictionary for all the log statements. The `aggFn` in `inflate()`is a user provided function used to process the log arguments programmatically, and the time/log id fields are processed outside `inflate()`.

After all the user sources have been processed, the NanoLog combiner is invoked to combine the contents of all the metadata files and generate a single dictionary source file for use with the NanoLog library and (optionally) the post-processor. It combines the map of identifiers to functions from multiple metadata files by trivially merging them and eliminating duplicates based on the whether the tags matches or not. It then canonically orders the map by performing a ASCII sort on the tags and assigns each log entry in the map a 4-byte integer, known as the unique log identifier, equal to the index of the entry's sorted tag. It uses this ordering to create an array of functions for the compress() and inflate() functions, assign values to the unique log identifiers in the system (i.e. the `extern` integers are assigned their tag's index in list of sorted tags), and generate a mega-writeDictionary() function that invokes the individual writeDictionary() functions in canonical order (Figure 4.9). The mega-writeDictionay() function is used to output the static information in the log file, and the array of functions is used by the runtime/post-processor to invoke the correct compress/inflate function based on the unique log identifier (a 4-byte integer).

4.3.6 Consistent Error Reporting

NanoLog's injected code can potentially cause the compiler to generate compile-time errors that are completely unintelligible for the user. Normally, when there is an error with some source code, the compiler will print a short description of the error followed by where it occurred in the source file (called *context*). The context information usually includes a source file, line number, a line offset, and a small snippet of the offending code (Figure 4.10a,c). This aids the user in quickly finding the offending line of code that caused the error. Unfortunately, since NanoLog injects source code into the user application, it's possible for the injected code to shift references in source code and cause the context to refer to a completely unrelated segment of code (Figure 4.10b,d). Furthermore, NanoLog's injected sources may also introduce compiler errors that only exist in the injected code. In this case, the compiler will reference code that isn't visible to the user, causing further confusion. Lastly, since NanoLog effectively defers the call to `printf` to a post-processor, it's possible for there to be problems in the format string that won't be caught until decompression time. Thus, NanoLog needs to implement mitigations to ensure that errors are reported early and properly to the user.

To ensure that the user receives intelligible error messages, NanoLog employs three strategies. It uses C-preprocessor directives to realign the source line numbers, it preserves the indentations of the original arguments to preserve offset references by the compiler, and it reports certain errors early in the preprocessor Python script before the compiler sees the modified sources.

Figure 4.7 shows an example of how the NanoLog preprocessor utilizes GNU C-preprocessor

(a) main.cc

```
#include "NanoLog.h"

main() {
  NANO_LOG(NOT_A_REAL_VARIABLE, "Hello World %d", rand());

  printf("This is a completely unrelated line of code");
}
```

(b) Modified main.cc

```
#include "NanoLog.h"

main() {
  // Notice the line shift and the odd indent
  random_record_function(
                  NOT_A_REAL_VARIABLE, "Hello World %d", rand());

  printf("This is a completely unrelated line of code");
}
```

(c) Expected Error

```
main.cc: In function 'int main(int, char**)':
main.cc:4:2: error: 'NOT_A_REAL_VARIABLE' was not declared in this scope
  NANO_LOG(NOT_A_REAL_VARIABLE, "Hello World %d", rand());
           ^~~~~~~~~~~~~~~~~~~
GNUmakefile:54: recipe for target 'main.o' failed
make: *** [main.o] Error 1
```

(d) Line Alignment Error

```
main.cc: In function 'int main(int, char**)':
main.cc:6:18: error: 'NOT_A_REAL_VARIABLE' was not declared in this scope
  printf("This is a completely unrelated line of code");
                  ^~~~~~~~~~~~~~~~~~~~
GNUmakefile:54: recipe for target 'main.o' failed
make: *** [main.o] Error 1
```

Figure 4.10: An example of how compiler errors may be misaligned when code is injected into an application. (a) shows the original source code and (b) shows the sources with the "NANO_LOG" function replaced. Notice that it added two new lines, and indented the "NOT_A_REAL_VARIABLE" identifier more. (c) shows the error that should have been shown, and (d) shows how the error is both pointing to the wrong line in the code, and at the wrong offset (i.e. it's indented more).

directives and indentions to preserve the offset references by the compiler. The GNU C-preprocessor allows the user to place *Line Control* directives to inform the compiler from whence a line of code comes [19]. These directives have the format of a "#" character followed by the line number and filename of the originating source file. The NanoLog preprocessor uses these directives to realign the user sources after some code has been injected. An example can be seen near the `main()` function in Figure 4.7 where after NanoLog preprocessor injected the `record()` function, it realigns the `main()` function to start on line three. These directives are inserted whenever NanoLog has injected code. It's worth noting that the NanoLog preprocessor also aligns the injected `record()` functions to "generatedCode.h". This helps to inform the user if there are bugs in the generated code. Also evident in Figure 4.7 is how the preprocessor system maintains the original indentation of the arguments to the NANO_LOG invocation (line 26). This is done in case there's an error in one of the user's arguments (such as an undefined variable). With the preserved indention, the compiler will properly highlight the error in the user's arguments.

Finally, there are some errors that must be caught early and reported by the NanoLog preprocessor itself. The most important of these are problems with format strings, which can cause the log files to be indecipherable by the post-processor. Normally, without NanoLog, the errors with the format string would be reported at runtime as the format strings get evaluated. However, since NanoLog defers formatting, the error may not appear until much later when the user decompresses the logs days/months/years after the execution. It would be unfortunate if the user lost years worth of data due to errors in the format string that can be caught at compile-time.

The NanoLog preprocessor checks for errors in the format string and disallows the use of the "`%n`" specifier (which NanoLog cannot support). A NANO_LOG's format string encodes the order and allowable types of the dynamic arguments. Mismatches in types can cause the preprocessor to generate the incorrect logic to process the arguments. To catch these types of errors, the NanoLog preprocessor generates record() functions that have the expected signature of the format string. For example, if a NANO_LOG statement contained the format string `"Hello world %p, %d, %lf"`, the NanoLog system would generate a function with the signature `record(int level, char* fmtString, void*, int, double)`. This signature matches the specifier types according to the printf-specifications[3] [6] and achieves two goals. First, it ensures that any implicit conversion (such as a float to a double) will occur before NanoLog attempts to save the argument to its internal buffers. Second, if the arguments provided by the user do not match the types encoded by the format string,

[3]The first two arguments are for the required severity level and format string

then compiler will report that the arguments do not match the `record()` function signature.

Lastly, NanoLog prevents the use of the "`%n`" specifier by explicitly checking the format string as the Python preprocessor executes. Normally, the "`%n`" specifier is used to return the current character offset of the log message. For example, `printf("%s %n", "Hello", number)`, would cause `number` to contain the value 7 after the line executes. However, since NanoLog does not format strings at runtime, it cannot produce this value at runtime. Thus, the `%n` specifier is specifically disallowed by the NanoLog preprocessor.

4.3.7 Summary

In summary, Preprocessor NanoLog is a variant of NanoLog that uses a Python script to preprocess and modify the user sources at compile-time. It interposes on the compilation process with a GNUmake macro, identifies log messages in the user sources, and generates four functions per log message. The record() function is injected into the user sources after the "`#include "NanoLog.h"` line and replaces the NANO_LOG invocation, and the compress(), inflate(), and writeDictionary() functions are placed into the NanoLog library for the runtime and post-processor to use. It uses metadata files to track log messages across compilation units and inserts spaces in the user code to maintain consistent error reporting.

4.4 C++17 NanoLog

C++17 NanoLog is an alternative version of NanoLog that uses C++17 metaprogramming features rather than a preprocessor to perform its front-end duties. It is easier to use as it no longer requires the user to integrate a preprocessor into their build pipeline, and the system is more similar to a traditional C++ library; users only need to `#include` a header and link against the NanoLog library. C++17 NanoLog is required to fulfill the two duties of building a dictionary to de-duplicate static log information and generating specialized, per-log functions to `record()` and `compress()` the remaining dynamic log arguments. It achieves these goals by using variadic templates to build specialized functions, `constexpr` compile-time evaluation to build dictionary entries at compile-time, and a runtime system to assign unique log identifiers to the entries. As implied by the name, C++17 NanoLog uses advanced C++17 features and only works with compliant applications and compilers.

4.4.1 A Quick Primer on C++11/14/17 Features

C++17 NanoLog uses fairly new[4] and esoteric features of the C++17 specification. To aid the reader in understanding the implementation and its limitations, this section presents a short primer on features relevant to C++17 NanoLog: the `constexpr` modifier and variadic templated functions. Note that the section assumes a basic understanding of C++ and some of the features presented are not exclusive to C++17.

The `constexpr` keyword was introduced in C++11 and was gradually improved through the evolution of C++17 to allow for simpler syntax. The modifier "`constexpr`" indicates that an expression is constant throughout the execution of the application and that the value can be computed at compile-time. When the modifier is applied to a variable, it means that the value of the variable is/must be known at compile-time. This distinction from regular variables allows the compiler to optimize the user application binary by simply substituting a literal value whenever the variable is used instead of computing it at runtime. The modifier can also be applied to a function to indicate that the function *can potentially* be run at compile-time if all the function parameters are known at compile-time (i.e. they are literals or other `constexpr` variables). However, the compiler does not enforce that the function must be executed at compile-time. If the compiler can't execute the function, it generates assembly as normal to execute the function at runtime.

To force compile-time execution of a `constexpr` function at compile-time, one can define a `constexpr` variable equal to the return of the `constexpr` function. An example of this is shown on line 35 of Figure 4.11 where a Fibonacci function is forced to execute. In this usage, the input parameters to the `constexpr` functions must be themselves constant expressions or literals, otherwise the compilation will fail with an error. For this reason, NanoLog requires the format string to be a literal type so that both the preprocessor and C++17 versions of NanoLog can analyze the string and build metadata structures at compile-time.

Constant expression functions can implement fairly complex tasks. For example, the `countVowels()` function in Figure 4.11 shows an example of how a string can be parsed at compile-time. The function resembles a fairly normal looking imperative function with `for` and `while` loops[5]. The only difference is that they cannot modify any global variables or function arguments; all results must be passed through the `return` statement.

[4]The C++17 specification was only recently available at the time of this publication.

[5]This syntax was only recently allowed in the C++17 standard. Older standards required a more functional-style where each function composed of exactly one return statement and loops had to be implemented with recursive calls.

```cpp
#include <cstdio>
#include <cstring>

// Calculates the Fibonacci number at position n
constexpr int
fib(int n)
{
    if (n <= 1)
        return n;

    return fib(n-1) + fib(n-2);
}

// Counts the number of vowels in a string
// Note the value of 'N' is automatically deduced by the compiler.
template<int N>
constexpr int
countVowels(const char (&fmt)[N])
{
    int sum = 0;
    for (int i = 0; i < N; ++i) {
        if (fmt[i] == 'a' || fmt[i] == 'A'
            || fmt[i] == 'e' || fmt[i] == 'E'
            || fmt[i] == 'i' || fmt[i] == 'I'
            || fmt[i] == 'o' || fmt[i] == 'O'
            || fmt[i] == 'u' || fmt[i] == 'U')
            ++sum;
    }
    return sum;
}

int main()
{
    constexpr char string[] = "the quick brown fox jumps over the lazy dog";
    constexpr int fib11 = fib(11);
    constexpr int vowels = countVowels(string);
    printf("Fib(11) = %d and the string '%s' has %d vowels\r\n",
                fib11, string, vowels);

    // Expected Output
    //
    // Fib(11) = 89 and the string 'the quick brown fox jumps over the lazy dog'
    // has 11 vowels
}
```

Figure 4.11: C++17-compliant application demonstrating the use of constant expressions to perform compile-time computation. In this application, the computation of fib(11) and countVowels(string) is performed by the compiler and the results are loaded as literals for the printf call at the bottom.

The other major features NanoLog uses are templates and variadic templates. Templates have been available since the introduction of the C++ language, and they allow for type substitution in data structures and functions, similar to Java generics. However, unlike generics, every instantiation of the template with a unique type will cause a copy of the class/function specialized to that type to be inserted into the binary. For example, if one uses the standard library's templated `std::max` function [4] and specialize it to integers in multiple places (i.e. `std::max<int>(a, b)`), then a max function specialized for integers only will be injected by the compiler into the binary. Multiple uses of `std::max<int>` will use the same function. However, if `std::max<long>` is invoked, then another function specialized on `long` is injected in the binary. This behavior is desirable in NanoLog as it helps create specialized functions and de-duplicate uses. More specifically, if one specialized the `record()`function on the log arguments, then a unique record function is generated for unique each combination of log statement arguments, and multiple log statements with the same argument signature will reuse the same function instance.

To generalize `record()`to more than one template type, NanoLog utilizes variadic templates. Variadic templates are a form of templates where an arbitrary number of types can be specified. An example of such a template in the C++ standard is `std::tuple<class... Types>`. This class can accept and operate on an arbitrary number of different types (i.e. `std::tuple<int, int>` or `std::tuple<float, void*, long>`). To operate on data passed into these variadic templated functions, one can use recursion and function overloading to peel off arguments from the list and process them one by one. Figure 4.12 shows an example of how a variadic templated function can be used to process an arbitrary number of arguments. The `template<typename T, typename... Ts> printTypes(T, Ts... rest)` function in the figure is a variadic, templated function; it allows an arbitrary number of arguments to be passed in, and it will print the ASCII representation of the type and value of each argument. The triple dots "..." enable the function to accept multiple types/arguments and treat them as a single variable. These dots are called *parameter packs* and the existence of a parameter pack in the template is what makes a function a templated variadic function [3]. There are two parameter packs in the printTypes function; one is in the template, "`typename... Ts`" and the other is in the argument list, "`Ts... rest`". The first describes the types of the arguments the function can take, and the second describes the arguments themselves. To process each element in the packs individually, the function uses type deduction [5] to peel off the first elements of the two parameter packs into the `T` and `head` variables. It then passes the two into the singular `printType<T>(head)` function which overloads to a specific function near the top of the figure and processes a single type and argument individually. Finally, the `printTypes` function recursively

```
#include <cstdio>

// The following three functions are overloaded to accept and print int's,
// long's, and double's. One would normally complete the set to cover the
// types of arguments accepted by printf, but this is an example.
void printType(int i) {
    printf("Integer: %d\r\n", i);
}

void printType(long i) {
    printf("Long: %ld\r\n", i);
}

void printType(double i) {
    printf("Double: %lf\r\n", i);
}

// Base case recursion: no arguments remaining
void printTypes() { }

// Variadic Template that accepts an unbounded number of arguments and
// recursively prints their values one by one.
template<typename T, typename... Ts>
void printTypes(T head, Ts... rest) {
    // Peel off and process the first argument
    printType<T>(head);

    // Recursively process the rest
    printTypes<Ts...>(rest...);
}

int main()
{
    printTypes(1, 2.0, 3l);
    // Expected output:
    //
    // Integer: 1
    // Double: 2.000000
    // Long: 3
}
```

Figure 4.12: C++11-compliant example demonstrating the use of variadic templates to process an unbounded number of arguments and specialize the logic depending on the type of the arguments. `printTypes` accepts an arbitrary number of arguments and prints the type and value for each of its arguments. It does this by recursively peeling off the first argument in the argument list, processing it with `printType`, and invoking itself with the rest of the arguments. NanoLog uses a similar technique to build its `record()` and `compress()` functions. Note the `printType` has only been overloaded to accept integers, longs, and doubles; to build a more complete version, one would have specialize `printType()` for every type accepted by printf[6].

calls itself with the remaining pack parameters `Ts` and `rest`. Each recursive call peels off yet another type and argument set until the packs are empty and the base case of `void printTypes()` is reached. In this fashion, the `printTypes` function can accept multiple arguments and process them individually.

Using these variadic templates, one can construct a function that can accept an arbitrary number of arguments with varying types, much like the original printf API. However, unlike the original API, the type information is also available at compile-time allowing us to specialize the behavior of the function at compile-time instead of parsing the format string and interpreting the arguments at runtime.

Lastly, one feature of legacy C++ worth reviewing is the "`static`" modifier. In C++, the `static` modifier indicates that an entity has a lifetime equal to the lifetime of the execution. In other words, when applied to a variable, the variable is allocated when the program execution starts and is never deallocated until the program ends. This modifier allows one to define variables that remain valid and usable for the entire life the application.

Furthermore, static variables can be defined within local scopes, such as a function. When this occurs, C++ creates a variable that both outlives the execution of the function and can only be accessed inside the function. This functionality allows functions (or more generally, scoped logic) to "remember" results from previous executions or perform memoization. NanoLog exploits this feature to assign a dictionary reference to the `record()` function and have the function "remember" it between invocations.

4.4.2 Generating Specialized Functions

C++17 NanoLog implements the `record()` and `compress()` functions as variadic templated functions. The goal of the generated functions is to in-line the logic to process arguments based on their type so that type deduction and format string parsing do not need to occur at runtime. Variadic templated functions fit this purpose perfectly. They can be used to force the compiler to recursively build functions specialized to process exactly the types of the arguments passed into the log function, similar to Figure 4.12. This removes the need to use a preprocessor to generate the functions. Furthermore, the recursive logic can be flattened into inline logic by compiling with any sort of GNU GCC optimization enabled. This removes all the intermediate and superfluous call/ret instructions in between recursive calls and creates functions that are nearly as optimized as the ones generated by Preprocessor NanoLog.

One problem with using the variadic templates alone is that they specialize exactly on the types passed into the function, not based on the types expected by the format string. This causes two problems: type mismatches are not caught and it's impossible to deduce the behavior for "`char*`" arguments based on types alone. Consider the statement `printf("Hello world %d")`. Notice that the printf requires an integer argument, but none is provided. Since the variadic templates only look at the argument types themselves to generate the functions, it would compile and execute without error. The problem would only be caught when the user attempts to decompress the log file and the post-processor attempts to use the missing argument. Additionally, if the functions encounter a `char*` argument, it's ambiguous whether the functionality should be to operate on the pointer value (in the case of a "`%p`" specifier) or the string contents (in the case of a "`%s`" specifier). These two problems are solved using `constexpr` functions and GNU attributes.

NanoLog leverages `constexpr` functions to parse the string literal format string at compile-time to disambiguate `char*` arguments. As mentioned in the C++17 Primer (Section 4.4.1), the `constexpr` modifier can be used to force functions to evaluate at compile-time if the function's arguments are known at compile-time. In NanoLog, the NANO_LOG statement requires a literal format string and that format string contains all the information needed to disambiguate `"%p"` vs. `"%s"` specifiers. Thus, we can utilize `constexpr` to build a binary array describing whether an argument at the n-th position should be interpreted as a string or pointer based on the format string. This is done in a fashion analogous to `countVowels` in Figure 4.11 where the format string is iterated through and the result of the type is returned (except a data structure is returned rather than a single integer). The record() and compress() functions can then utilize this data structure to reduce the disambiguation logic to a single if-check of the data structure. This *is* extra logic that needs to be executed, but fortunately it only needs to occur for "`char*`" arguments.

Lastly, to prevent mismatched types, NanoLog tags the `record()` function with the GNU "format" attribute. This attribute is specific to the GNU GCC compiler and it instructs the compiler to statically check that the number and types of arguments passed into a formatting function match a format string. If the types do not match, then a compile-time error is generated.

With constant expression functions and variadic templates, NanoLog is able to build functions that are specialized to process exactly the arguments of a specific NANO_LOG statement without runtime analysis of the format string. The two functions generated, `record()` and `compress()`, can then be blindly invoked by the application logging thread and the NanoLog background thread.

4.4.3 Building the Dictionary at Runtime

In addition to generating optimized functions, the front-end is also responsible for maintaining a dictionary to de-duplicate the static log information and track when log messages are invoked for the first time. C++17 NanoLog builds the dictionary entries at compile-time, and the entries are assigned unique log identifiers and collated in the log at runtime. Unlike Preprocessor NanoLog, C++17 NanoLog cannot perform arbitrary computation, accesses, and modifications on the source files at compile-time, especially across multiple source files. Instead, it is limited to some compile-time computation through constant expression evaluations which *must* be localized to where the data is defined since the constant expression functions need direct access to its literal parameters[6]. As a result, C++17 NanoLog can only build the dictionary entries at compile-time in the local scope and must pass the entries to the NanoLog runtime. This means extra runtime compute is needed to support C++17 NanoLog relative to preprocessor NanoLog.

4.4.4 Building The Dictionary Entry

The C++17 front-end builds dictionary entries that resemble Figure 4.13 using constant expression evaluation. Each entry contains fields for the static log information (filename, line number, severity, format string), and three variables needed by the runtime. The first variable needed by the runtime is the *compressFnPtr*; it stores a pointer to the generated `compress()` function and is invoked by the background thread to compress the dynamic log data. The other two variables describe the types of the dynamic arguments as specified by the format string. One is the number of dynamic arguments, and the second is an array of enumerations informing the runtime whether an argument in a particular argument position is a string or a pointer. The array, function pointer, and number of dynamic arguments are built at compile-time with the use of constant expressions and there is one dictionary entry per log statement in the source.

One oddity worth mentioning is the compression function's specialization with variadic templates in Figure 4.13. Typically, template types are used to refer to the types of the arguments accepted by the function. However, this doesn't need to be the case, and it's not the way it is used in NanoLog's compression function. In NanoLog's case, the variadic templates inform the compiler which specialized functions to inline into the `compress()` function. The function always accepts

[6]In C++17, literal types and constant expression variables cannot be passed up the stack and remain classified as constant expressions.

```
// Defined in the C++17 NanoLog library to store static log information
struct DictionaryEntry {
    // Stores the traditional static log information
    const char* filename;
    const int linenum;
    const int severity;
    const char* formatString;

    // Used by the NanoLog background thread to compress the dynamic args
    void (*compressFnPtr)(int, const bool*, char**, char**);

    // Stores a precomputed array that describes the type encoded for
    // each parameter. Primarily used to distinguish %s from %p.
    const ParamType* parameterTypes;
    const int numParameters;
}

...

/**
 * Extremely simplified compress function from C++17 NanoLog. It accepts an
 * input byte array and compresses the log entries to an output byte array.
 * The template parameters are used to encode what the function needs to do
 * in order to interpret the input byte stream.
 *
 * \tparam T      - Type of the head argument (i.e. the current one) to compress
 * \tparam Ts     - Types of rest of the arguments to compress
 * \param argNum - The current argument we're processing (zero based)
 * \param params - Array of enums identifying the types
 * \param input  - Input array pointer to read log data from
 * \param output - Output array pointer to write compressed log data to.
 */
template<typename T, typename... Ts>
inline void
compress(int argNum, const ParamType *params, char **input, char **output)
{
    compressSingle<T>(paramTypes[argNum], input, output);
    compress<Ts...>(argNum + 1, params, input, output);
}
```

Figure 4.13: A simplified version of the dictionary entry structure used in C++17 (top) followed by the compression function (bottom). On the top, the `parameterTypes` field is an array of enumerations that encode the types of the dynamic arguments as specified by the format string. It is used used by the compression function to disambiguate `const char*` dynamic arguments.

four arguments: a counter, the `parameterTypes` array, an input buffer, and an output buffer. However, to interpret the types in the input buffer, the function needs to know the types of dynamic arguments encoded in the buffer and in which order. This information is passed to the function via the variadic template pack parameter. The function can then use the recursion techniques talked about earlier to peel off a single template type argument from the pack and invoke a specialized `compressSingle<T>` function. The `compressSingle<T>` function is specialized to read back exactly one argument of type `T` from the input array, compress it, and write it to the output array. Since the types are templated and known at compile-time, the compiler can unroll the `compressSingle` functions and generate in-line code. The only place where this technique falls short is for string and pointer types (i.e. `const char*`); in this case the function dereferences `parameterTypes` at runtime to determine whether to save the pointer's value or string contents[7].

The use of variadic templates and constant expression evaluations allows the C++17 front-end to build a specialized `compress()` functions as well as the dictionary entries at compile-time.

4.4.5 Collecting the Entries at Runtime

The C++17 runtime collects the dictionary entries at runtime via a `registerInvocationSite` function. This function is invoked when a log statement is executed for the first time and it adds the dictionary entry to the runtime dictionary (more details in Chapter 5). It then returns a unique log identifier that will be used by the runtime and post-processor to refer to the static log information. The function is fairly expensive as it requires a lock to serialize dictionary operations, so the C++17 log function amortizes this cost; it saves the return value into a local-scope static variable (`logId`) and uses the variable as a fast-check so that future invocations of the same log statement need not re-execute the function.

As a side note, the fast-check on `logId` is only a heuristic. Due to parallelism in modern processors, it possible for multiple threads executing the same log message to see a stale value for `logId` and invoke the register function multiple times. This behavior is suboptimal from a performance standpoint, but it is also expected to be short lived. Eventually the processor will fetch the latest value for `logId`, and all invocations from then on would prevent multiple executions. The `registerInvocationSite` function is also engineered to be safe. It accepts the `logId` variable by reference, and takes a dictionary lock before re-checking the variable and assigning to it. The lock

[7]As a reminder, the `record()` function uses the same technique.

ensures that the variable truly is uninitialized before adding the static log information to the dictionary. The heuristic is not perfect, but it is both safe and performant in the log run.

4.4.6 The C++17 NANO_LOG function

The C++17 NANO_LOG function is what allows the specialized functions to be generated and dictionary entries to be built. The C++17 log "function" is actually a C-style macro that expands to a new scope (Figure 4.14 and Figure 4.15). This expansion allows the C++17 system to define variables to build a dictionary entry at compile-time via constant expressions and define a local-scoped static integer, `logId`, to store the unique log identifier assigned by `registerInvocationSite`. After the dictionary entry is collected, the macro invokes the generated `record()` function to place the dynamic arguments into a staging buffer.

4.5 Preprocessor vs. C++ 17 NanoLog

The C++17 version of NanoLog uses C++17's strong compile-time computation support to analyze the log statements and generate optimized code and data structures for the runtime. The advantage of such a scheme is that it does not require an additional source preprocessing step and can therefore be easily integrated into any project that uses a C++17-compliant compiler. Additionally, since it doesn't inject functions directly into the library sources, the NanoLog library can be compiled once and reused in multiple projects like a normal library. This is unlike the Preprocessor version which requires the library to be rebuilt on every compilation in order to incorporate the specialized code generated by the preprocessor.

C++17 NanoLog does have three shortcomings when compared to Preprocessor NanoLog: it must collect the dictionary entries at runtime instead at at compile-time, the functions it generates are less optimized, and error reporting is more difficult.

The first shortcoming of C++17 NanoLog is that it cannot collect all the static log information into a dictionary at compile-time. This means that the static log information must be collected at runtime and output incrementally as new log messages are encountered. Although some of the analysis on the log statements can be done at compile-time, the assignment of unique identifiers must occur at runtime. This incurs a slight cost at runtime when a new log message is encountered

C++17 NANO_LOG Definition

```
 1  /**
 2   * NANO_LOG macro used for logging in C++17 NanoLog.
 3   *
 4   * \param severity - The LogLevel of the log invocation
 5   * \param format   - The printf-like format string
 6   * \param ...      - Log arguments associated with the printf-like string.
 7   */
 8  #define NANO_LOG(severity, format, ...) do {                                \
 9                                                                              \
10      /* Build the constant expression portions of the dictionary. */        \
11      /* Note: countFmtParams and analyzeFormatString are omitted for space*/ \
12      constexpr int nParams = countFmtParams(format);                         \
13      static constexpr std::array<ParamType, nParams> paramTypes =            \
14                              analyzeFormatString<nParams>(format);           \
15                                                                              \
16      /* Associates a dictionary entry to this log statement's scope */       \
17      static int logId = UNASSIGNED_LOGID;                                     \
18                                                                              \
19      if (severity < NanoLog::getLogLevel())                                   \
20          break;                                                               \
21                                                                              \
22      /* Triggers the GNU printf checker to check for format errors */         \
23      if (false) { checkFormat(format, ##__VA_ARGS__); }                      \
24                                                                              \
25      NanoLogInternal::log(logId, __FILE__, __LINE__, severity, format,       \
26                              paramTypes, ##__VA_ARGS__);                     \
27  } while(0)
```

Figure 4.14: A simplified version of the NANO_LOG macro used in C++17 NanoLog; the full source is available in the NanoLog Github repository[70]. Demonstrates C++17 NanoLog's use of NANO_LOG as a macro to define a new scope (`do... while(0)` in lines 8-27), constant expression evaluations on the format string to build portions of the dictionary entry (lines 12-14), declaration of the scoped static variable (line 17) used to "remember" the unique log identifier assigned by the dictionary. The specialization of the `compress()`function and assignment of the static, unique log identifier is performed in the `log(...)` function invocation on line (25), and the function is shown in Figure 4.15.

C++17 NANO_LOG Definition - Part 2

```
 1 /**
 2  * Log function invoked by the C++17 NANO_LOG macro to register the dictionary
 3  * information and record the arguments. This logic exists in a separate
 4  * function to make it more readable.
 5  *
 6  * \tparam N     - Number of print arguments (automatically deduced)
 7  * \tparam M     - Static length of the format string (automatically deduced)
 8  * \tparam Ts... - Types of the log arguments passed in (automatically deduced)
 9  *
10  * \param logId      - Reference to the static, unique log identifier integer
11  * \param file       - Filename of the where the NANO_LOG statement is invoked
12  * \param line       - Line number of where the NANO_LOG statement is invoked
13  * \param severity   - Severity of the NANO_LOG statement
14  * \param format     - Format string of the NANO_LOG statement
15  * \param ParamTypes - Describes the argument types encoded in the format string
16  * \param Ts...      - Parameter pack containing all the log arguments
17  */
18 template<long unsigned int N, int M, typename... Ts>
19 inline void
20 log(int &logId,
21     const char *file,
22     const int line,
23     const LogLevel severity,
24     const char (&format)[M],
25     const std::array<paramType, N>& paramTypes,
26     Ts... args)
27 {
28     // Fast-check for whether the system needs to add a new dictionary entry
29     if (logId == UNASSIGNED_LOGID)
30     {
31         // compress() is specialized here on the template arguments <Ts...>
32         DictionaryEntry entry = {
33                                   file, line, severity, format,
34                                   sizeof...(Ts),  paramTypes.data(),
35                                   &compress<Ts...>
36                                 };
37         // logId is passed by reference; its value is filled in by the function
38         RuntimeLogger::registerInvocationSite(entry, &logId);
39     }
40     ...
41     record<Ts>(...);
42 }
```

Figure 4.15: This is a continuation of Figure 4.14, and shows the `log()` function invoked by the NANO_LOG macro. The static, unique log identifier (`logId`) is passed in by reference on line 20, checked on line 29, and assigned on line 38. The `compress()` function is specialized by passing the template parameter `Ts...` on line 35, the dictionary entry is added to the runtime on line 38, and finally `record()` is specialized and invoked on line 41.

for the first time. However for long running tasks, we expect the dictionary cost to be amortized[8].

Another shortcoming is that the functions generated by C++17 NanoLog can be less optimized than the ones generated by Preprocessor NanoLog. As a reminder, C++17 NanoLog uses a combination of variadic templated functions to specialize on the argument types and constant expression evaluations to derive a data structure disambiguating "`%s`" vs. "`%p`" specifiers from the format string. For some reason[9], the compiler disallows the use of constant expressions derived from a literal format string to be used as a non-type template parameter. This essentially means that the `record()` and `compress()` functions cannot be specialized to the data structure, and the data structure must instead be passed in as a regular function parameter and checked at runtime. This both incurs a runtime cost and limits the amount of optimization that can be performed by the compiler.

It does appear that the compiler is able to optimize the data structure checks out of the `record()` logic, but it is unable to do so for the `compress()` logic. I suspect the reason is because the `record()` function is invoked immediately after the definition of the derived data structure. This allows the compiler to "see" the values of the data structure at compile-time as it inlines the `record()` functionality. On the other hand, since `compress()` is invoked in a separate thread and in a separate scope, the compiler cannot optimize the function as easily.

Finally, C++17 NanoLog can produce compiler errors that are harder to decipher by the developer using NanoLog. Since NanoLog defers formatting until a later point, it needs to catch as many potential user errors as possible at compile time. Otherwise, errors that would normally be caught at runtime may slip past the user until they decompress the logs later on. The Preprocessor version of NanoLog has the luxury to arbitrarily examine the source code and report the error in a contextually relevant way. In C++17, the errors are reported by the compiler, which can have notoriously difficult to decipher error messages when templates are involved. Figure 4.16 shows samples of the error reports generated by Preprocessor and C++17 NanoLog when an invalid format specifier is used.

[8]Furthermore, C++17's runtime collection of static log information means that if a log message is never used, then it'll never have its dictionary entry cataloged. This saves a small amount of space in the log file.

[9]At least, *I* was unable to figure out a way to incorporate the information derived from the literal format string into a form that can be consumed as a non-type template parameter.

Preprocessor NanoLog Error

```
g++ -E -I ../runtime  main.cc -o  main.cc.i -std=c++11
python ../preprocessor/parser.py --mapOutput="generated/ main.cc.map"  main.cc.i

main.cc:65: Error - Unrecognized Format Specifier: "%q"

    NANO_LOG(NOTICE, "Sample %q", 5);

GNUmakefile:54: recipe for target ’main.o’ failed
```

C++17 Error Reporting

```
g++ -I ../runtime -c -o main.o main.cc -std=c++17 -DNDEBUG -O3 -g
In file included from main.cc:24:0:
../runtime/NanoLogCpp17.h: In function ’int main(int, char**)’:
main.cc:68:5:   in constexpr expansion of
    ’NanoLogInternal::getNumNibblesNeeded<10ul>("Sample %q")’
../runtime/NanoLogCpp17.h:334:39:   in constexpr expansion of
    ’NanoLogInternal::countFmtParams<10>(fmt)’
../runtime/NanoLogCpp17.h:311:24:   in constexpr expansion of
    ’NanoLogInternal::getParamInfo<10>(fmt, count)’
../runtime/NanoLogCpp17.h:204:68: error: expression ’<throw-expression>’ is not
    a constant-expression
                             "Unrecognized format specifier after %");
                                                                    ^
...
In file included from main.cc:24:0:
../runtime/NanoLogCpp17.h:285:1: note: candidate: template<int NParams, long
    unsigned int N> constexpr std::array<NanoLogInternal::ParamType, NParams>
    NanoLogInternal::analyzeFormatString(const char (&)[N])
 analyzeFormatString(const char (&fmt)[N])
 ^~~~~~~~~~~~~~~~~~~
../runtime/NanoLogCpp17.h:285:1: note:   template argument
    deduction/substitution failed:
In file included from main.cc:24:0:
../runtime/NanoLogCpp17.h:1083:66: error: no matching function for call to
    ’log(int&, const char [8], int, NanoLog::LogLevels::LogLevel, const char
    [10], const int&, const int&, int)’
                             numNibbles, paramTypes, ##__VA_ARGS__); \\
                                                                  ^
main.cc:68:5: note: in expansion of macro ’NANO_LOG’
     NANO_LOG(NOTICE, "Sample %q", 5);
     ^~~~~~~~
GNUmakefile:23: recipe for target ’main.o’ failed
make: *** [main.o] Error 1
```

Figure 4.16: Shows the errors reported by Preprocessor NanoLog (top) and C++17 NanoLog (bottom) when an invalid specifier is used in the format string. The format string that caused the error was `"Sample %q"` where `"%q"` was the invalid specifier. The C++17 error report has been truncated to fit within the page boundaries.

4.6 Summary

The NanoLog front-end is the compile-time component of the NanoLog system, and prepares the user's application for optimized logging. It reduces logging I/O by cataloging static log information, persisting it once in a dictionary, and changing the log statements to emit only dynamic information. It also reduces computation in the application by generating `record()` and `compress()` functions that are specialized to process exactly one log statement in the sources. The former allows the application to reduce the amount of static, never-changing information in the log file, and the latter allows the runtime to execute optimized, in-lined code at runtime. Both prepare the application for nanosecond scale operations.

There are two variants of the front-end, Preprocessor and C++17. The preprocessor version uses a Python script to scan and modify the user sources at compile-time, and the C++17 version uses strong meta-programing features to force the C++ compiler to generate optimized code and build dictionary structures. The difference between the two is that the C++17 version is more usable, while the preprocessor version is both more performant and displays errors in a more user-friendly format. Conceptually, both versions of the NanoLog front-end perform the same tasks, and either can be used to prepare the application for nanosecond scale logging.

Chapter 5

Runtime

The NanoLog runtime is a statically linked library that runs as a part of the user application. Its primary purpose is to support the optimizations leveraged by the NanoLog front-end and produce a runtime log file that is interoperable with the post-processor. It is responsible for maintaining and persisting the dictionary of static information produced by the front-end, and for ferrying the remaining dynamic log arguments to disk in an efficient manner.

In this chapter, I will discuss how the runtime manages the dictionary, how it encodes dynamic log data, and how it is optimized for nanosecond scale operations.

5.1 Managing The Dictionary

NanoLog utilizes a dictionary to deduplicate the static log information in the log file. The system separates the repeating, unchanging static information (such as the filename, line number, severity, and format string) from the constantly changing dynamic information (such as the time of invocation and format arguments) of a log message and persists them separately (Figure 5.1). The static information is written in the dictionary and persisted just once, while the ever-changing dynamic information is persisted with every log invocation and includes a reference (*unique log identifier*) to the dictionary of the static information.

The dictionary can be thought of as a simple array of dictionary entries. Each entry contains a single log message's filename, line number, severity level, and format string. There is exactly one dictionary entry for every log message in the application's sources, and the index of each entry is the application-wide *unique log identifier* for that log statement. It is stored separate from the dynamic log data as it never changes and thus can be written to the log file just once, rather than with every

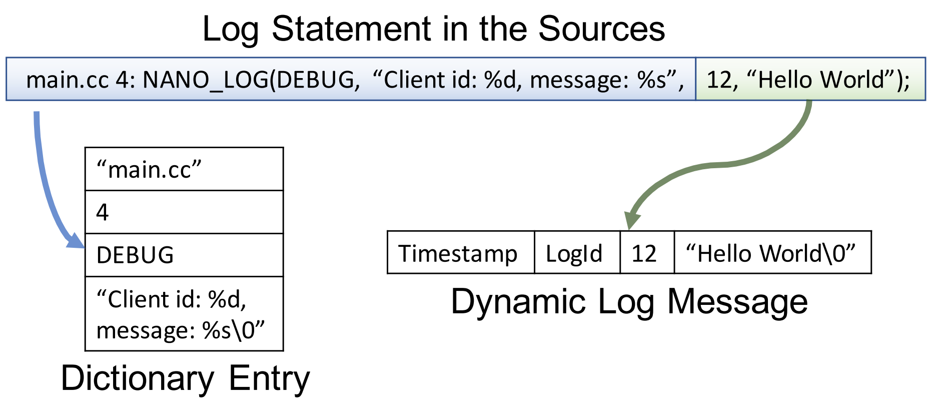

Log Statement in the Sources
main.cc 4: NANO_LOG(DEBUG, "Client id: %d, message: %s", 12, "Hello World");
"main.cc"
4
DEBUG
"Client id: %d, message: %s\0"
Dictionary Entry
Timestamp
LogId
12
"Hello World\0"
Dynamic Log Message

Figure 5.1: An example of a log statement in the NanoLog system and the resulting dictionary entry (bottom left) and dynamic log message (right). The dictionary entry on the left contains the log message's filename, line number, severity, and format string, and the dynamic log message contains a 64-bit timestamp counter (Intel TSC [29]), the unique log identifier (logId), and the two arguments for the log statement. The dictionary entry is persisted only at most once per log file, while the right entry is persisted on every log message invocation.

log message invocation.

The only constraint is that the dictionary entry for each log statement must be written to the log file before the first reference to it by the dynamic log information. The post-processor requires both the static (dictionary) and dynamic information of a log statement to process it. Thus, the runtime ensures that the dictionary entry is written in the log file before the first reference to it by the dynamic log data. This way, the post-processor will have encountered all the information it needs to decode a log statement when it reaches the dynamic log data.

The NanoLog runtime uses different techniques to manage and persist the dictionary depending on which version of NanoLog is used (Figure 5.2). For preprocessor NanoLog, almost all the work is done by the front-end component. Preprocessor NanoLog catalogs all the static log information at compile-time and injects a `writeDictionary()` function into the binary that dumps the dictionary information to the log file. The information output by this function is both complete and exhaustive (i.e. it covers every log message in the system). Thus, the runtime simply needs to invoke this function to insert the dictionary at the beginning of every new log file, and the post-processor is guaranteed to encounter the dictionary entries before first use.

In C++17 NanoLog, the dictionary management is more complicated than the preprocessor version (Figure 5.2). The runtime component needs to track when new log statements are encountered, collect the static log information at runtime, and output the dictionary entries before first use. To

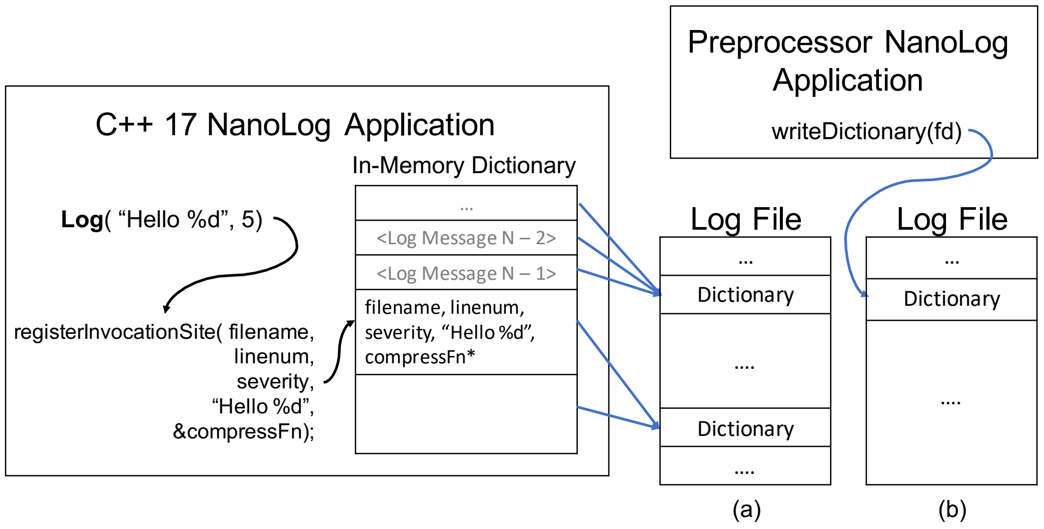

Preprocessor NanoLog Application
writeDictionary(fd)
C++ 17 NanoLog Application
In-Memory Dictionary
Log("Hello %d", 5)
...
<Log Message N – 2>
<Log Message N – 1>
registerInvocationSite(filename, linenum, severity, "Hello %d", &compressFn);
filename, linenum, severity, "Hello %d", compressFn*
Log File
...
Dictionary
....
Dictionary
....
Log File
...
Dictionary
....
(a)
(b)

Figure 5.2: Dictionary management schemes in C++17 NanoLog (left) and preprocessor NanoLog (right). In preprocessor NanoLog, the application is provided with a `writeDictionary()` function that will dump the entire dictionary into the log file. This function is compiled into the application by the front-end, and is only executed once per log file (note the single arrow to the log file). In C++17 NanoLog, dictionary is collated at runtime. The application thread invokes "Log", which triggers "registerInvocationSite" to store the static log information in an in-memory dictionary. This dictionary is then output piece-wise into the log file (note the multiple dictionary fragments in the log file). The "registerInvocationSite" function is guaranteed to add only one dictionary entry per log message (see Chapter 4).

accomplish the first two tasks, the C++17 front-end uses memoization with static integers to ensure that a function, `registerInvocationSite()`, is invoked at most once per NANO_LOG statement. This function informs the runtime that a new log statement is encountered and passes the static information to the runtime system. The runtime then creates a new dictionary entry for the static log information, pushes it onto a shared vector of dictionary entries, and returns the index of the new entry as the unique log identifier[1]. The log identifier can then by used by the dynamic log data to refer to the dictionary, and the vector of dictionary entries is periodically written to the log file.

In C++17 NanoLog, the dictionary is written in a piece-meal fashion. The runtime will periodically check the shared vector of dictionary entries and output new ones to the log file between other log data. This results in a log file that resembles the Figure 5.2a, where the dictionary is split into fragments throughout the log file. Additionally, since the data structure is shared, it's possible for other threads to add entries while the runtime is processing log data. To ensure that dictionary entries are written to the log file before first reference, the C++17 runtime additionally tracks which dictionary entries have been written to the log file. When it encounters dynamic log data that does not have its corresponding dictionary entry written yet, it will first write the dictionary entry before continuing to process the dynamic log data. In this manner, the runtime ensures that all dictionary entries will exist in the log file before their first reference.

Overall, the NanoLog system has different ways to manage and output the dictionary depending on which version of NanoLog is used, but it will always ensure that the dictionary entries are output before the dynamic log information refers to it.

5.1.1 Structure of the Dictionary

The NanoLog dictionary maps an integer identifier to static log information.

In memory, the dictionary is implemented as a simple array of dictionary entries. Each entry contains the filename, line number, severity level, and format string of a log statement, and the array index of each entry determines the unique log identifier used by the dynamic information to reference the static log information.

The runtime representation of the dictionary also includes an array of `compress()` functions generated by the front-end. These functions are used by the runtime to compress the dynamic log data prior to output, and are indexed in the same manner as the dictionary (i.e. the unique log identifier

[1]This design does imply that the unique log identifiers, and hence the dictionary, can be different for each execution of the application depending on which log statements execute first.

encoded in the dynamic log data is used to dereference the corresponding `compress()` function). There exists exactly one function for every log message in the source. This part of the dictionary is not persisted to disk, as it is only needed to optimize the runtime's operations.

On disk, the dictionary is represented as a series of dictionary fragments. Each fragment can be thought of as a contiguous chunk of the runtime dictionary array, and can trivially be appended together in the order they're encountered to form the full dictionary. Each fragment starts with a small header indicating its byte size and the number of dictionary entries encoded in the fragment. Following the header are individual dictionary entries; each one contains a log statement's severity in integer form, the line number, the byte lengths of the filename and format string, followed by the filename and format string. The dictionary entries are written in the same order as they are in the runtime implementation, so the post-processor can trivially collect the dictionary fragments, append them together, and form the dictionary required to decode the dynamic log data. The preprocessor version of NanoLog outputs the entire dictionary in one fragment, while the C++17 version outputs mulitple fragments.

One important note about the dictionary is that every log file needs a copy. The NanoLog system allows users to set a new destination for the log data and split the log file at runtime. When this happens, the dictionary needs to be copied into the new destination to ensure that every log file can be decompressed independently. For preprocessor NanoLog, this means that the `writeDictionary()` function is invoked for every file change. For C++17 NanoLog, this means it must iterate through its internal dictionary array and output all the static information collected since the beginning of execution with each new log file. Despite the way it sounds, C++17 NanoLog will always output an equal or lesser amount of information than its preprocessor counterpart. This is because log messages that are never executed will not have a corresponding dictionary entry in the C++17 runtime.

5.2 Encoding Log Messages

The NanoLog system persists only the dynamic log information for log message invocations. In a traditional logging system, log messages are output in the full-human-readable format at runtime. A typical log message would include the time of invocation, filename/line number of the log message, the thread identifier, severity, level, and the formatted message itself. In the NanoLog system, the static information is filed away in a dictionary and formatting is deferred. This allows the runtime system to only output the minimal set of dynamic information along with a reference into the dictionary for the post-processor to use. This makes the log messages extremely compact and more efficient to output (as it eschews formatting at runtime). Figure 5.1 shows how a sample log message in the source is split into its static and dynamic components.

Log messages output by the NanoLog system only contain a timestamp, a reference to the dictionary, and the dynamic format arguments passed into NANO_LOG. The timestamp identifies when the log message was created, the reference to the dictionary identifies which log statement was invoked, and the dynamic arguments specify the values which the user wished to format. The invocation time is a copy of the Intel Timestamp Counter [29] when the log statement is invoked and is stored as a delta relative to the last previous message. The integer reference is a four-byte index into the dictionary, and the log arguments are written in the order in which they appear in the original log statement's argument list. String arguments are written with null-terminators to indicate length, and non-string arguments are encoded as their native in-memory representations. See the bottom right of Figure 5.1 for an example of the dynamic log message representation. Note that a log message's thread identifier is not included in the representation for each log message; this is because messages in the log file are grouped by threads and the grouping shares a single identifier.

Upon output onto disk, the dynamic log message representations above are grouped into batches called *buffer extents*. Each buffer extent is a contiguous chunk of log message produced by a single thread in the application's execution. An extent additionally encodes the length of the extent, the thread identifier that produced the chunk (i.e. which thread invoked the log messages at runtime), and the log messages themselves. Since all the log messages within an extent are produced by a single thread, they are naturally ordered by time.

One feature of the NanoLog system is that it outputs the dynamic log arguments without delimiters between them. In a strawman approach to encoding log messages, one may be tempted to insert type identifiers between log message's arguments so that the post-processor knows the byte length and how to process each argument. However, this mechanism would incur additional storage overheads for each log message, so NanoLog does not use it. Instead, NanoLog leverages the invariant that format strings can never change in the system and encodes the dynamic log information in the order specified by the original format string. This way, the post-processor can utilize the format string stored in the dictionary to infer argument type and order, and explicit delimiters and type identifiers between arguments are not needed. For example, the format string "`Client id:%d, message:%s`" in Figure 5.1 would imply that the dynamic log data contains an integer and a null-terminated string as the dynamic arguments.

Lastly, the log messages themselves are compressed, however the mechanism will not be discussed here. Instead, the full description of the mechanism along with an evaluation of its efficacy can be found in Chapter 8.

5.3 Structure of the Log File

Applying the techniques above, the NanoLog runtime produces log files that resemble Figure 5.3.

The first component in each log file is the header. It contains a map between the machine's Intel Timestamp Counter [46] (TSC) and wall time along with a conversion factor between the two. The header allows the log messages to contain the raw TSC values and avoids wall time conversion at runtime, and the post-processor can use the mapping and conversion factor to perform this task at post-execution.

Following the header is the first dictionary fragment. Each fragment contains dictionary entries that map an integer to a log message's static information, i.e. the filename, line number, severity, and format string for a log message. There is at most one entry for every log message in the original source[2], and the entries are written in array order; i.e. their ordering in the log file determines the unique log identifier that maps to them. Additionally, multiple dictionary fragments within a log file can be trivially appended with one another to form a larger dictionary. In preprocessor NanoLog there is only a single, large fragment at the beginning of each new log file whereas in C++17 NanoLog, additional dictionary fragments can appear later in the log file. The post-processor is expected to collect and concatenate the fragments together to form the dictionary required for interpreting the log messages.

Following the first dictionary fragment are buffer extents. Each *buffer extent* encodes a contiguous chunk of log messages produced by a single thread at runtime. The log messages in each extent are naturally chronologically ordered since they originate from the same thread, but log messages between extents can overlap in time as they can originate from separate runtime thread. Chapter 6 will describe how the post-processor sorts the log messages between extents to reform the full, chronologically ordered human-readable log file.

The rest of the log file can contain additional headers, dictionary fragments, and buffer extents in any order. Additional headers can be inserted in the log file to realign the TSC and wall times if clock skew is detected. Additional dictionary fragments can be included to extend the dictionary when new log messages are encountered. And finally, additional buffer extents will be written as additional logging statements are executed.

[2]C++17 can have no entries for a log message if it's never executed at runtime.

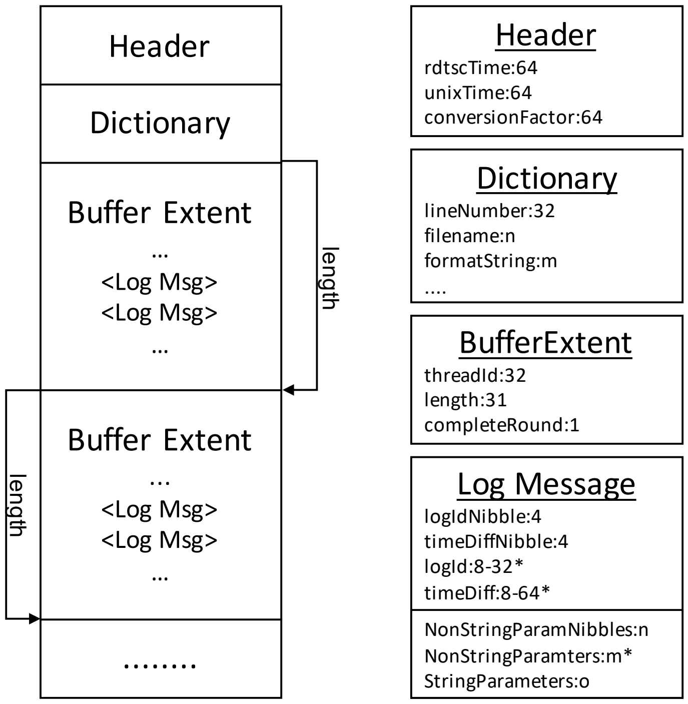


Figure 5.3: Layout of a binary log file produced by the NanoLog runtime at a high level (left) and in more detail(right). As indicated by the diagram on the left, the NanoLog output file always starts with a header, a dictionary fragment, and one buffer extent. The rest of the file can contain additional headers, dictionary fragments, and buffer extents in any order. On the right, each component is expanded. The header contains a mapping of wall time to the machine's timestamp counter with a conversion factor between the two. The dictionary fragment contains entries of static log message information (i.e. the messages' filenames, line numbers, severities, format strings), and the buffer extents contain log message encoded by a single thread. On the right, the smaller text indicates field names and the digits after the colon indicate how many bits are required to represent the field. An asterisk (*) represents an integer value that has been compressed and thus has a variable byte length. The lower box of "Log Message" represent dynamic arguments that are variable length (and sometimes omitted) depending on the log message's arguments. Preprocessor NanoLog has the dictionary encoded in full after the header, while C++17 NanoLog encodes dictionary fragments throughout the log file.

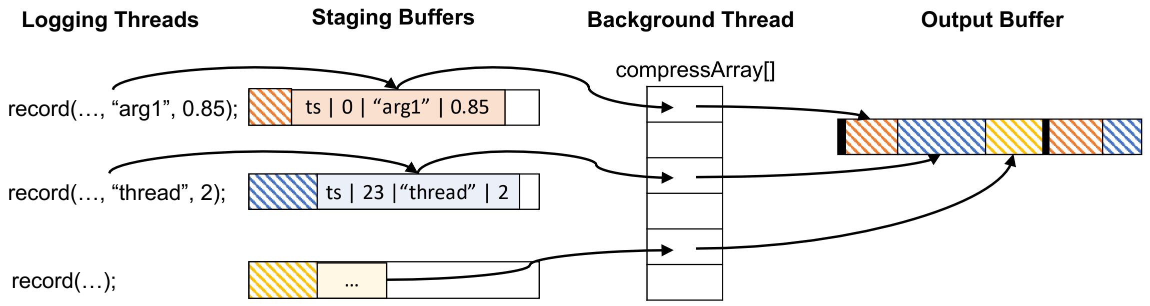


Figure 5.4: Overview of the interactions that occur with the NanoLog runtime system. On the left, the user application initiates by executing the generated `record()` function for a log statement. This copies the log statement's timestamps (ts), unique identifier, and dynamic arguments into a thread-local staging buffer. At a later point, the background thread polls the staging buffers and invokes the `compress()` function associated with each log statement in the staging buffers. The compress function places the output in the output buffer (to the right). The logging threads execute in parallel, while the background thread is single threaded and processes the staging buffers in a round-robin fashion. The different colored stripes in the right diagram represent chunks of data originating from different staging buffers.

5.4 NanoLog Runtime Architecture

Now that we understand how the NanoLog system manages and writes the dictionary and dynamic log data, let's look at how the runtime is implemented.

The NanoLog runtime is a static library that is compiled into the user application. It serves to buffer the log data produced by the application logging threads and writes them to disk. It provides low-latency, in-memory *staging buffers* to accept the logging threads' dynamic log data, and a background thread to collect the data and prepare them for more efficient, batched I/O.

Figure 5.4 shows the operation of the runtime component. It starts when an application thread performs a logging operation. It invokes the front-end generated `record()` function and places the log statement's dynamic log arguments into a staging buffer. There is one staging buffer per application thread to allow for concurrent operation. The NanoLog background thread will then periodically wake up, poll the staging buffers for data, and invoke the front-end generated `compress()` on each log statement to prepare it for output. The results of `compress()` are stored in an intermediate output buffer and the buffer is flushed when it is full or no messages have been logged for a predetermined amount of time.

5.4.1 Enabling Nanosecond Scale Operations

The design of the NanoLog runtime evolved around the need for performance, and it leverages four key optimizations to enable its nanosecond scale operations.

First, it uses precomputed logic provided by the front-end. The `record()` and `compress()` functions used to place and compress data in the staging buffers are pre-generated by the front-end. They contain straight-line code that operates exactly on the data present in the buffer. Using these functions eliminates the need for the runtime to semantically parse the format string and understand the log arguments to operate on them. Instead, it invokes these functions which already know the layout of data and what transformations to perform.

Both the `record()` and `compress()` functions are highly specialized to exactly one log statement in the source, and care must be taken to ensure that the correct function is invoked. The correct `record()` function is always invoked as the front-end replaces the NANO_LOG statement in the source with an invocation to the `record()` function instead. The corresponding `compress()` function is encoded in the dictionary[3] and can be dereferenced with the log message's unique log identifier (which is persisted by `record()` as a part of the dynamic log data). Overall, these two functions reduce the cost of logging by eliminating branching compute and executing logic specialized exactly to the message being logged.

Second, the runtime *staging buffers* are optimized for nanosecond scale operations by avoiding contention and synchronization wherever possible. The runtime uses a series of in-memory buffers, called the *staging buffers*, to store the output of the low-latency application threads, and decouple them from the high latency operation (such as disk I/O). These buffers are optimized to avoid both language level and machine level synchronization (i.e. locks and caches). A full discussion of the design and optimizations of the staging buffers is presented in Chapter 7.

Third, the background thread processes the log messages in contiguous chunks. In a traditional logging platform, the system is required to output the log messages in a single, chronological ordering. This requires either all the logging threads to serialize on a central data structure, or it requires a background process to access multiple buffers to sort the log messages (and thus incur more cache misses). Both solutions require extra compute at runtime. The NanoLog system is different; it defers formatting and is allowed to output the log messages at runtime in whichever ordering is most convenient. Thus, the background thread optimizes for data locality. It will scan through the staging

[3]In C++17 the compress function is placed into the dictionary via the registerInvocationSite function, and in preprocessor NanoLog the compress function array is compiled into the runtime.

buffers one at at a time and for each, consume as much log data as possible before moving on to the next staging buffer. This optimizes for data locality as the background thread processes contiguous chunks of memory, rather than issuing reads to multiple buffers in order to sort the messages. The trade-off with this technique is that the sorting will now have to be done at post-processing, which will be discussed in Chapter 6.

Finally, the runtime employs compression to enable more efficient I/O. In order to maintain nanosecond scale operations, the sink (i.e. disk) must be fast enough to drain the staging buffers and keep them from filling. Thus, the background thread uses a form of compression, called *variable integer encoding*, to reduce the I/O time. I will defer the discussion of the compression system, as the full design and evaluation of the compression subsystem is described in Chapter 8. However, I've come to the conclusion that this form of compression is most efficient for the types of data that NanoLog produces.

5.5 Summary

The NanoLog runtime is the intermediary between the application and post-processor; it must accept log messages by the application and encode them in a format understandable by the post-processor. This responsibility includes cataloging the static log information into a dictionary (C++17 only), outputting the dictionary, and encoding the log messages in a minimal, but sufficient format for the post-processor to understand.

The NanoLog runtime also is a performance critical component of the NanoLog system; it sits in the hot-path between the application performing logging operations and the high latency disk operations. To perform well, it utilizes precomputed logic from the front-end (Chapter 4) to `record()` and `compress()` log statements, it implements low-latency, lockless staging buffers (Chapter 7) to buffer log data, outputs log data in contiguous chunks for the post-processor to sort (Chapter 6), and employs compression (Chapter 8) to reduce the I/O size.

Chapter 6

Post-Processor

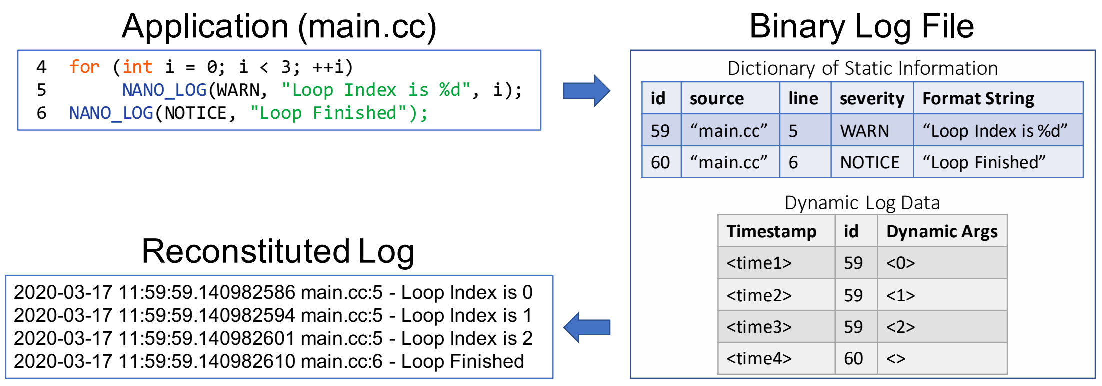
The post-processor is the final component in the NanoLog system. It is responsible for interpreting the binary log file produced by the application and reconstructing the original log statements for either human or robot consumption.

6.1 Reconstructing the Log File

The reason for the post-processor's existence is because NanoLog's log file is not directly consumable for most users. Unlike traditional logging platforms, the NanoLog system does not output complete log messages in a human-readable format at runtime. Instead, it outputs the log file in a binary format, and it breaks up a log statement's static and dynamic components and persists them in separate parts of the log file for performance. The static, never changing information (i.e. filename, line number, severity, and format string) is cataloged into a dictionary and written at most once in the log file, while the dynamic information is persisted with every log invocation and includes a reference (*unique log identifier*) to the dictionary of the static information. Figure 6.1 shows a simplified example of the encoding. Thus to reconstruct the complete message, the post-processor must rebuild the dictionary of static log information for lookups and use the reference within the dynamic information to retrieve the static information. These two pieces of information together form the complete log message.

6.1.1 Reconstructing the Dictionary

In the NanoLog system, the dictionary maps a unique log identifier (an integer) to a log message's static log information. It is implemented as an array of dictionary entries where each entry contains a



```
4  for (int i = 0; i < 3; ++i)
5       NANO_LOG(WARN, "Loop Index is %d", i);
6  NANO_LOG(NOTICE, "Loop Finished");
```

Dictionary of Static Information

id	source	line	severity	Format String
59	"main.cc"	5	WARN	"Loop Index is %d"
60	"main.cc"	6	NOTICE	"Loop Finished"

Dynamic Log Data

Timestamp	id	Dynamic Args
<time1>	59	<0>
<time2>	59	<1>
<time3>	59	<2>
<time4>	60	<>

```
2020-03-17 11:59:59.140982586 main.cc:5 - Loop Index is 0
2020-03-17 11:59:59.140982594 main.cc:5 - Loop Index is 1
2020-03-17 11:59:59.140982601 main.cc:5 - Loop Index is 2
2020-03-17 11:59:59.140982610 main.cc:6 - Loop Finished
```

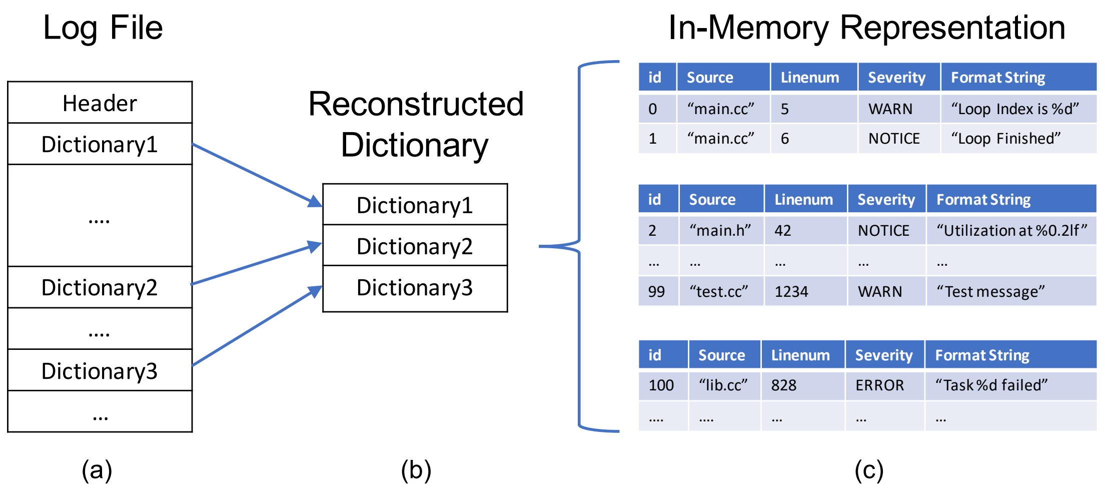
Figure 6.1: Demonstrates how the NanoLog system splits log messages (top left) into static and dynamic components and persists them separately in the log file (right). The static log information (such as the source file, line number, severity, and format string) is persisted just once in the dictionary, while the dynamic information (such as the timestamp and dynamic arguments) is persisted for each invocation of a log statement. The dynamic information also saves a reference to the dictionary (id) which can later be used to recombine the separate components into the full log message (bottom left).

log statement's filename, line number, severity, and format string, and the index of entry determines the unique identifier that maps to it. There is at most one dictionary entry for each log statement in the sources and it is needed by the post-processor to reconstitute the complete log message.

To reconstruct the original dictionary, the post-processor collects dictionary fragments in the log file and concatenates them together (Figure 6.2). The runtime encodes the dictionary as a series of one or more fragments in the log file. Each fragment represents a contiguous chunk of the dictionary array with one or more dictionary entries. The order in which the entries are written reflects their ordering in the runtime dictionary. Thus, to reconstruct the dictionary, the post-processor only needs to collect and concatenate the dictionary fragments.

Furthermore, the runtime ensures that dictionary entries are written in the log file before the first reference by the dynamic log information. This means at any point in the log file, the dictionary is considered complete (i.e. covers all dynamic references until the next dictionary fragment) as long as all prior fragments have been collected.

Once rebuilt, the dictionary can be thought of as a lookup table (Figure 6.2c). Each entry in the table contains a log message's static information and is indexed by a log message's unique log identifier (encoded with the dynamic log information). With the dictionary, the post-processor can decode the dynamic information and form the complete message.



id	Source	Linenum	Severity	Format String
0	"main.cc"	5	WARN	"Loop Index is %d"
1	"main.cc"	6	NOTICE	"Loop Finished"

id	Source	Linenum	Severity	Format String
2	"main.h"	42	NOTICE	"Utilization at %0.2lf"
...	...	...	...	...
99	"test.cc"	1234	WARN	"Test message"

id	Source	Linenum	Severity	Format String
100	"lib.cc"	828	ERROR	"Task %d failed"
....	....	...	...	...

Figure 6.2: Shows how the post-processor (a) collects the dictionary fragments in a log file and (b) concatenates them together to form the reconstructed dictionary. (c) shows the contents of the dictionary fragments from (a) and (b). Note how the id's (unique log identifiers) for the dictionary entries in the fragments are sequential; allowing them to be trivially concatenated together to form the full dictionary.

6.1.2 Reconstructing the Log Message

The goal of the post-processor is to reconstruct complete messages for the user or another application to consume. It can either produce the full, human-readable log messages that other logging platforms produce (similar to what's shown in the bottom left of Figure 6.1) or present the data via an application programming interface. To perform either task however, it must reunite the log statement's dynamic information with its static information. The following section will describe how this is done in the context of producing the full, human-readable log statement.

The encoding NanoLog uses for the dynamic portion of the log message is extremely minimal. It only consists of an 8-byte timestamp, a 4-byte unique log identifier, and the dynamic arguments encoded in their native C++ in-memory representations without delimiters or headers in between the arguments[1] (Figure 6.1).

What's missing from this representation is the log message's context information and how to parse/format the dynamic arguments. The context information is typically needed to produce the

[1]In practice, compression is applied to both the header and the dynamic arguments, however this detail is omitted in this discussion. See Chapter 8 for more details on compression.

final, human-readable log message and consists of a log statement's filename, line number, and severity. It helps to orient the user in identifying the log statements in the source file, and is simply missing in this representation. Another piece of information that's missing is how to parse the dynamic arguments. Since the arguments are encoded without delimiters or headers, it's impossible to know where one argument ends and another begins without additional information. Finally, the representation is missing the format string, which tells the post-processor how to produce the full-human readable message.

Fortunately, the missing information can be found by dereferencing the dictionary of static information built in the previous section. By using the unique log identifier encoded in the dynamic portion of the log message as an index, post-processor can retrieve the log statement's context information and format string from the dictionary. The context information can be used to format the context portion of the human-readable string, and the format string both informs the post-processor on how to parse the dynamic arguments and produce the full-human readable message. More specifically, the post-processor uses the ordering of the format specifiers encoded in the format string [6] to infer the type and order of the dynamic arguments. For example, a format string containing `"%d"` such as in Figure 6.1 indicates that there is exactly one integer-width argument to parse and format. Another example is `"%ld %0.2lf %s %p"`, which indicates that there is a long, float, string, pointer encoded in the dynamic arguments in that order. The post-processor can then use this information parse the arguments and read them into memory.

At this point, the post-processor has all the information it needs to reform the original log statement. It has the static log information (via the dictionary), the time of invocation, and a means of parsing the dynamic arguments into in-memory representations. If the goal is to reproduce a human-readable log message, then the system would convert the time of invocation into a "YYYY-MM-DD HH:MM:SS.ns" format, output the context information (i.e. the filename, line number severity), parse the dynamic arguments according to the specifiers in the format string, and pass the arguments along with the format string into a final `printf()` statement to form the full message. If the goal is to expose the log message via an API for another application to consume, then it would buffer the static and dynamic information into a data structure and allow the application to query the arguments (more on this later). The reconstruction of the log message is considered complete at this point.

To summarize, the full order of operations to reform a single log statement is as follows: The post-processor first starts reading the log file from the beginning. When it encounters a dictionary fragment, it reads the dictionary fragment into memory and stores it as an array of dictionary entries. If more fragments are encountered, their entries are trivially append to the array of dictionary entries. This forms the working dictionary. When the post-processor encounters a log statement, it uses the unique log identifier as a key into the dictionary and retrieves the static information. The format string from the static information is then used to infer the type and ordering of the dynamic arguments, and the post-processor reads the arguments into an in-memory data structure. At this point, the post-processor has all the information it needs to reconstruct the original log message and can either output it in a human-readable format or present them to another application via an API.

6.1.3 Amortizing the Cost of Parsing the Format String

The operation of parsing the format string to interpret the dynamic arguments is amortized in the NanoLog post-processor. The design in the previous section implies that the format string needs to be repeatedly parsed in order to infer the order and types of the dynamic arguments. This operation is expensive as the parsing code contains many branches and iterates through every character in the format string. Fortunately, since the format strings never change in the NanoLog system, it is sufficient to parse the format string once and build an auxiliary structure to describe the order and type information. This data structure can then be used in lieu of the format string to interpret the dynamic arguments.

The format specifiers in the format string are key to understanding types and order of the dynamic arguments. They follow the prototype of "%[flags][width][.precision][length]specifier" where the length and specifier together indicate the type of the dynamic argument to be used [6]. When placed in a format string, the sequence of specifiers determines the types and order of the dynamic arguments. For example, "`%d %10.2s %lf`" indicates that the dynamic arguments should encode an `int`, `char*`, and `double` in that order. Thus to eliminate the repetitive parsing of the format string, the post-processor needs to represent the information encoded in the specifier in a more machine-friendly way.

The post-processor amortizes the cost of parsing the format string by building an auxiliary structure to describe it. It splits format strings into smaller *format fragments* that contain at most one format specifier each and tags each fragment with an enumeration describing the type of the dynamic argument expected by this fragment. The concatenation of all the fragments results in the original format string, and the order of the fragments in the array reflects the order of format specifiers in the original format string. This array of fragments is then saved into the dictionary alongside the context information and format string. When parsing the dynamic arguments, the post-processor can then simply lookup this data structure in the dictionary, iterate through the fragments, and use the enumerations to determine the types of the arguments encoded. This avoids the need to repeatedly parse the format string.

One detail omitted above is the specification of dynamic width and precision parameters in the format specifiers. The NanoLog API is modeled after the printf API [6], and the printf API allows users to specify asterisks (`"*"`) for the width and precision parameters in the format specifier. In these situations, the user is indicating that an additional dynamic value for the width and/or precision is desired and will be passed in via the arguments list before the actual format argument. For example, a specifier of `"%*.*lf"` indicates that two integers and a double will be passed into the argument list; the first two integers are the dynamic width and precision parameters and the double is the actual argument. To account for these extra arguments, each format fragment additionally includes two booleans to indicate whether dynamic width and/or precision values have been used. If so, the post-processor will read back the two integers before reading the argument itself.

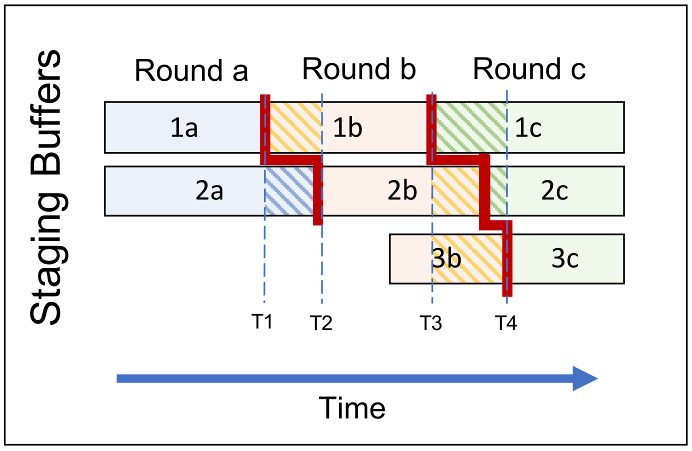
In summary, the post-processor splits each format string in the system into an array of format fragments. Each fragment contains a substring of the original format string with at most one format specifier and is tagged with an enumeration describing the type of argument expected and two booleans to indicate dynamic width/precision. When processing a dynamic log statement, the post-processor will lookup this array in the dictionary, and iterate through the fragments to determine the order and type of the arguments (along with the need for additional dynamic width/precision values). This amortizes the cost of parsing the format string to just once per dictionary entry.

6.1.4 Sorting the Log Messages

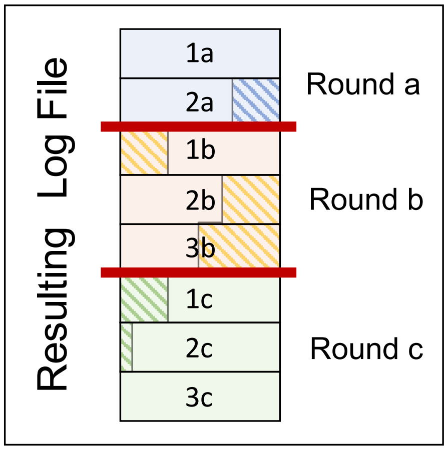
The sections above describe how the post-processor reconstructs a single log message in a log file. This section describe how the post-processor ensures proper chronological order before outputting the log statements.

Simply processing the log statements in the order they appear in the log file will generally not produce a time-ordered output in the NanoLog system[2]. Recall from Chapter 5 that the runtime background thread does not output the log messages in chronological order at runtime. Instead, the runtime simply iterates through the staging buffers in round-robin order and attempts to output

[2]Unless the application only had a single active logging thread at runtime. In this case, all messages will be naturally ordered since they're only produced by one thread.



(a)



(b)

Figure 6.3: NanoLog outputs buffer extents in rounds. (a) shows the runtime configuration and (b) shows the resulting log file. In (a), each row represents a different staging buffer (1, 2, 3) storing the log data produced by a particular thread and the horizontal axis represents time. Labels T1 and T3 indicate times when the background thread starts reading log data from the staging buffers and encoding them into buffer extents (i.e. they indicate the start of an output round), and labels T2 and T4 correspond to when the background finishes an output round through all the staging buffers for T1 and T3 respectively. (b) shows the resulting log file with three rounds of output. In between the times when the background thread is active ([T1, T2] and [T3, T4]), log messages produced by the logging threads can appear either in the current output round (N) or next round (N+1). These messages are highlighted via hashes in blue, yellow, and green to indicate their uncertainty in (a) and (b). Finally, the jagged red lines that cuts through (a) indicate the actual time position when the background thread stopped reading a particular staging buffer before moving onto the next. These red lines delineate when an output round starts and ends for each buffer. They are represented as horizontal lines in (b) to indicate the limits of the “round” labels.

the entire contents of a staging buffer into a *buffer extent* before moving onto the next. When the background thread completes a transit through all the staging buffers, we call this an *output round* and the runtime marks when this logically occurs with a bit in the buffer extent header. This results in a fragmented log file similar Figure 6.3b, where log messages are time-ordered within a buffer extent, but not between extents. If the post-processor were to process the log statements in file order, it would see chunks of log messages that are ordered relative to each other, but jumps in time between log messages whenever a new buffer extent is encountered.

A strawman approach to ordering the log statements would be to read all the buffer extents into memory and perform a sort operation between all the messages. This solution can be improved by treating the problem as an external sort problem [18]. Conceptually, the concatenation of all buffer extents produced by a single runtime thread forms a sorted sublist of log messages in the system. This is trivially true since all the log messages are produced by the same thread. Thus, the post-processor can buffer at least one buffer extent per runtime thread at a time and perform the merge operation from external sort. Whenever a buffer extent is emptied of log messages, the post-processor would scan ahead in the log file, find the next extent belonging to the same thread, buffer it, and continue merging.

However, both strawman solutions suffer from unbounded buffering or unbounded scanning through the file. The first solution requires the entire log file to be read into memory, and the second solution requires the post-processor to always be able to find the "next" buffer extent for a particular runtime thread. If a thread is quiescent for a long period of time, the "next" buffer extent may be very far in the file. This would require the post-processor to scan arbitrarily far in the log file to find the "next" buffer extent.

Fortunately, we can bound the amount of buffering to just two rounds of output for the merge sort technique. To understand why, let us consider how the runtime operates. At runtime, each logging thread is allowed to produce log messages in parallel and store them in staging buffers independent of each other. At some point in time, say T1 from Figure 6.3, the background thread will start an output round, read the buffers one-by-one, and write their complete contents to disk as buffer extents. Since the buffer extents within a output round originate from different threads, they must be buffered and merge sorted. This explains why buffering at least one round is necessary. At some point in the future, say T2, the background thread will finish encoding the output round and go to sleep. Now since the logging threads are allowed to add new log messages while the background thread is reading the contents of the staging buffer, there is a period of uncertainty between T1 and T2, depending on which staging buffer the background thread is currently processing, where a log message from that time period can either appear in the current round (round a in Figure 6.3) or the next round (round b). Thus, the post-processor must buffer at least two output rounds and perform merge-sort. The reason why a third round (c) is not required, is because the second round (b) must have started encoding at some time T3 $>$ T2, which means that all the log data generated during the uncertainty period (T1 $\leq$ x $\leq$ T2) will be captured in that second round. Thus, the post-processor only needs to buffer at least two rounds to perform a successful merge sort.

Admittedly, there can be one extremely rare scenario in which two rounds of output may not be sufficient. This situation occurs when a thread executing `record()` is preempted after it reads the log message's timestamp counter, but before it finishes saving the arguments to the staging buffer. In this case, it's possible for this thread to pause for greater than two rounds of output before finally persisting the rest of the log message to the staging buffer and allowing the background thread to process it. This could result in the background thread outputting a log message with a timestamp significantly earlier than any log other log message in the previous output round. However, this scenario is extremely unlikely in practice for two reasons. First, the amount of code between reading the timestamp and persisting the arguments is extremely short and contains no branches, so it is unlikely to be preempted at this point. Second, the problem can only occur if the thread is preempted and stays descheduled for more than two disk round-trips, two periods of NanoLog background thread wakeups, and two rounds of compression. This is a long period of time that the thread is unlikely to stay descheduled for. I have inserted assertions in the post-processor that will trigger and throw an error if this scenario is encountered. However despite multiplexing thousands of threads onto 4-8 core machines, the assertions have never triggered. So in practice, the event occurs so rarely that buffering two output rounds is sufficient[3].

6.2 Directly Consuming the Binary Log

One of the most interesting aspects of the post-processor is the promise it holds for faster analytics. Most analytics engines have to gather human-readable logs, parse the log messages into a binary format, and then compute on the data. Almost all the time is spent reading and parsing the log file. Directly consuming the NanoLog log file can significantly speed this operation in two ways: first, it saves I/O by consuming a smaller, compressed log file, and second, it forgoes the expensive formatting and ASCII parsing of a traditional analytics engine. Taken together, it's possible to perform

[3] Although the GitHub implementation of the post-processor [70] buffers three output rounds to be additionally safe.

```
Decoder decoder;
decoder.open("logFile");

int count = 0;
int sum = 0;

// Assume there is exactly one log message that matches our search
int targetMsgId = decoder.getLogIdsContainingSubstring("Student \%d scored \%d points out
    of 100.")[0];

LogMessage msg;
while(getNextLogStatement(msg)) {
  if (msg.getLogId() == targetMsgId) {
    int score = msg.get<int>(2);
    sum += score;
    ++count;
  }
}

printf("The average score was %d", sum/count);
```

Figure 6.4: Sample application using the NanoLog aggregation API to perform a mean aggregation on the second argument of log messages starting with "Student %d scored %d points out of 100." One simplification made in the code above is the `getLogIdsContainingSubstring()` function; it does not exist in the implementation of NanoLog API (as of the writing of this dissertation). Instead, there exists an equivalent CLI tool to dump all the log identifiers containing a certain substring into the terminal, and the user is expected to manually inspect the list and assign `targetMsgId` to the correct value in their application.

aggregations 1-2 orders of magnitude faster than conventional methods (See Chapter 9: Evaluation). As a proof of concept, the NanoLog post-processor offers an analytics API to consume log files in a binary format.

The post-processor aggregation API allows users to iterate through the log file message by message and query the static and dynamic arguments. Figure 6.4 show a sample application using the post-processor's API to perform a mean aggregation on the second argument of a specific log statement in the log file. The user first creates a `Decoder` object, `open()`'s the file, and iterates through the log file via `getNextLogStatement()`. While the post-processor is able to get a new log statement, the user is given a `LogMessage` object from which they can query the unique log identifier and the dynamic arguments via `getLogId()` and `template<typename T> T get()` respectively. The `get` function is templated as the arguments can have any basic type and C++ disallows functions with multiple return types in any other circumstance. This does mean the user must know the type of the argument they want to access. The user can also query the log message's timestamp, filename, line number, and severity of the log statement via separate functions, but that is not shown in the

"Pure" Preprocessor inflate() API

```
inline void
inflate(char *in, FILE *outputFd void (*aggregate)(...))
{
  // Static info directly embedded in source
  const char *fileName  = "TableManager.cc"
  const int lineNum     = 1031;
  const char *logLevel  = formatLogLevel(NOTICE);
  const char *fmtString = "Creating table '%s' with id %d";

  // Dynamic information extraction
  int id                = unpack<int>(in);
  const char *time      = formatTime(unpackTime(in));
  const char *arg1      = unpackString(in);
  int arg2              = unpack<int>(in);

  ...

  // At this point we have all information, can either
  // print the log message or call an aggregation method.
  if (aggregate != NULL) {
    *aggregate(fmtString, arg1, arg2);
  } else {
    fprintf(outputFd, "%s %s:%d %s: ", time, fileName, lineNum, logLevel);
    fprintf(outputFd, fmtString, arg1, arg2);
  }
}
```

Figure 6.5: Sample `inflate()`code generated by "Pure" Preprocessor NanoLog for the log statement `"Creating table '%s' with id %d"` in file "TableManager.cc" on line 1031. The function in-lines the static information and the extraction of the dynamic information directly in code, and it passes the information into an `aggregate()` function. The function accepts the same parameters as the original NANO_LOG statement.

example.

Overall, the API is fairly rudimentary at the moment. It primarily serves as a proof-of-concept to show the performance gains of such an API. Future users of NanoLog are expected to implement a more expressive system.

6.3 "Pure" Preprocessor NanoLog

In this chapter thus far, I have only discussed an implementation of the post-processor that uses a dictionary encoded in the log file (common to both C++17 and Preprocessor NanoLog and used by default). There exists another version of the post-processor that avoids encoding a dictionary in the log file altogether. This version of the post-processor is specific to preprocessor NanoLog only, and it uses the front-end to compile the dictionary directly into the post-processor application. This version of the post-processor is now deprecated as it's more difficult to maintain, however it will be discussed here for posterity.

This version of the post-processor relies on the preprocessor to generate and inject the functions necessary to interpret the dynamic log statements directly into the post-processor application. For each log statement in the source, the preprocessor generates an `inflate()` function that both reads back the dynamic data encoded in the binary log file and formats the final log message. The static information that's normally encoded in the dictionary is instead embedded as logic directly in the function itself. For example in Figure 6.5, the context information (such as the filename, line number, severity, and format string) is directly inserted into the source as local variables and the logic to parse and format each dynamic argument is in-lined. The `inflate()` functions are then placed into an array and compiled into the post-processor. The post-processor can then use the unique log identifier encoded in each log statement as a lookup index to dereference the array and invoke the correct `inflate()` function per log message.

This design leads to a more performant post-processor. The logic to interpret the dynamic data is directly inlined into the sources. It does not require the post-processor to reconstruct the dictionary, nor parse the format string to interpret the dynamic arguments. Instead, everything is encoded as straight, in-lined logic. This both reduces the number of branching statements to be executed at post-execution (as parsing is no longer required), and allows the compiler to more aggressively optimize the function (as the functions are straight, in-lined code).

Furthermore, this design changes the API to consume the log messages programatically. Instead of being handed a `LogMessage` object to retrieve the static/dynamic log data, the user is expected to pass an aggregation function pointer to the `inflate()` function. This function is invoked after all the dynamic arguments have been parsed (as shown near the bottom of Figure 6.5) and is expected to accept the same arguments in the same order as the original NANO_LOG statement. Note, the other static information is retrieved outside the `inflate()` function via a dictionary lookup table, and the static information embedded in the function is for the `fprintf` invocation.

However, this version of the post-processor is deprecated[4] due to two usability issues. First, it does not work with the C++17 implementation of NanoLog, as it relies on a preprocessor to inject logic into the post-processor. Second, this design requires users to maintain specific versions of the post-processor for specific versions of the application. Since the dictionary of information is

[4]Though it is still accessible on the NanoLog GitHub page [70].

locked into the post-processor at compile-time, only a specific post-processor can be used with a specific compilation of application. If the post-processor is lost or the wrong one is used, the log file becomes indecipherable. Thus, the dictionary variant of the post-processor is preferred as it is compatible with C++17 NanoLog and allows users to use a generic post-processor for all NanoLog applications.

6.4 Summary

The NanoLog post-processor is the final component in the NanoLog pipeline. It consumes the binary log files produced by the runtime application and presents the log data for either human or robot consumption. In operation, it rebuilds the dictionary of static log information, parses the dynamic log data, and either formats the final message in a human-readable format or presents it via an aggregation API. The post-processor optimizes the parsing of the dynamic log data by building an auxiliary data structure to describe the format specifiers in format string, and it leverages how the runtime encodes log data to limit the amount of buffering used for sorting. Finally, the post-processor offers an aggregation API to consume the log messages programatically, and it can optionally have the dictionary compiled directly into itself for performance in “pure” preprocessor NanoLog.

Chapter 7

The Staging Buffers: A Cache Conscious Design

The NanoLog staging buffers exist as part of the runtime component (Chapter 5) and serve to buffer application log data before consumption by the background I/O thread. Due to their position in the NanoLog system, the design of the staging buffers must be simultaneously low latency to support the logging threads' nanosecond scale operations and high throughput to support optimized, batched I/O by the background thread.

In this chapter, I will discuss the various techniques and optimizations used by the staging buffers to achieve nanosecond scale operations. This discussion will start with a strawman implementation that suffers micro- to millisecond delays then improve it in several steps to an implementation that can provide operations on the scale of single digit nanoseconds.

7.1 The Problem/Setup

The core function of the staging buffers is to decouple the logging threads' low-latency operations from the high-latency ones, such as disk I/O. The NanoLog system is optimized to produce extremely minimal log data at extremely low latency. The messages are on the order of tens of bytes and the log operation can complete in under ten nanoseconds per message. Disk devices, on the other hand, are optimized for large I/O. Disk drives take on the order of microseconds to milliseconds to seek and only achieve their highest throughput with large I/O's. Thus, there is a mismatch in performance. If each logging thread were to directly write log data to disk with each log invocation, then each operation would have to wait on the order of milliseconds for disk writes to complete. To bridge this performance gap, NanoLog uses buffering.

The NanoLog system uses in-memory buffers, called *staging buffers*, and a background thread to collect many small log messages and group them into one large I/O. In this design, each application thread writes the log data to an in-memory buffer, and a background thread will poll the buffer, collect the log entries, and write them in large batches to disk. This design amortizes the cost of the disk seek operation and effectively allows the disk device to write at a higher speed. Furthermore, the use of in-memory buffers allows the logging thread to perform very lean operations; it only needs to push log data into a queue before returning. The overheads of organizing the log data, applying compression, and performing the disk I/O are all offloaded to the background thread. Together, these two mechanisms can allow the logging thread to operate in the nanosecond scale.

Since the in-memory buffers directly interact with the application threads, the implementation details of the buffers directly affects logging performance as seen by the user. The rest of this chapter will describe the techniques used by NanoLog's staging buffers to minimize log latency.

7.2 The Strawman: A Monitor-Style Queue

The simplest design for the buffer is a single, monitor-style queue. In this implementation, there is a centralized queue that all threads use, and the queue is protected with a single lock that controls access to the queue. Each application logging thread and the background thread must acquire the lock to read and write data to the buffer, and they must release the lock after use.

The benefit of this implementation is that is simple. Most modern programming languages include locking mechanisms and queue-like data structures. Thus to implement this version of the staging buffers, the implementer only needs to import the system libraries for the lock and queue, and instantiate global references for the logging and background threads to use.

This implementation also simplifies the runtime and post-processor. Since all the logging operations are serialized with one another, the log messages produced are naturally ordered by time. This simplifies the runtime and the post-processor as neither will need to sort the log messages before consumption by the user (more on this in Chapter 6: Post-Processor).

The downsides to such a simple implementation are contention, serialized access, and locking delay. In this design, all logging threads must serialize access to a single data structure. This limits the maximum performance of the system to that of a single logging thread, even when multiple cores are available. Additionally, the monitor-style lock implies that the background thread cannot operate in parallel with the logging threads; it cannot consume log data already written in the buffer while the logging threads add new data. This further slows down the logging threads as they must not only wait for each other, but also the background thread to complete its operations.

Furthermore, the locking operations provided by most languages are not appropriate for NanoLog's nanosecond scale operations. The log operation in the NanoLog system can complete in as little as seven nanoseconds, but a lock operation can take several times longer as it requires explicit and excessive cache synchronization. Even worse, certain implementations of language-level locks can put waiting threads to sleep. This can cause the logging threads to experience milliseconds of delay as the operating system reschedules the thread, even though the logging operation only requires nanoseconds to complete. Thus, locks can artificially slow down the system by several orders of magnitude.

7.3 Allocating Private Buffers

NanoLog's staging buffers attempt to avoid the problems associated with the strawman design. It allocates a separate buffer per logging thread, avoids the use of locks, and allows for concurrent access to the data structure between the logging and background threads.

NanoLog allocates a private staging buffer per logging thread instead of sharing a global data structure between the threads. This design avoids the need to synchronize access across logging threads and allows them to place log data into the buffers in parallel. The buffers are allocated either when the thread invokes a logging statement for the first time, or when the application explicitly invokes the `NanoLog::preallocate()` function[1]. The buffers are destroyed after the associated thread exits and all buffered log statements have been written to disk. The benefit of this design is that it allows for concurrent logging operations, but the downsides are that it requires more memory allocations/deallocations (at most one per thread creation/destruction) and the log messages are no longer ordered between threads.

One problem introduced with this design is that the natural, chronological ordering of the log statements is lost. In the strawman design, since all threads are serialized to a global queue, all log messages are naturally ordered by time. In this design, the logging threads are allowed to write log statements in parallel into separate buffers. Each buffer will maintain a chronological ordering of log statements, but messages between buffers are not necessarily ordered. Thus, additional sorting

[1]This function is to allow the user to control when the allocation occurs to prevent it from overlapping with performance critical log statements.

work needs to be performed at some point to re-order the log messages. In NanoLog, this problem is solved by deferring sorting to the post-processor. This is discussed in further detail in Chapter 6.

7.4 Lock-Free and Concurrent

An issue I have not addressed so far is that even with private staging buffers, the logging threads can still contend with the background thread. In particular, the background thread could lock the data structure and prevent the logging thread from placing data, even when there is free space in the buffer. Thus, the NanoLog staging buffers are also implemented as lock-free and concurrent data structures.

The NanoLog staging buffers are also lock-free and allow for concurrent operations. In this design, a logging thread is allowed to place data into its respective staging buffer while the background thread is reading data already written to it. This design improves throughput as the logging thread is no longer stalled by the background thread and vice versa. To achieve this result, the staging buffers carefully order memory operations to ensure that invalid data is never read, and they segregate state variables so that some are only written to by the logging thread and others are only written to by the background thread.

The NanoLog staging buffers allow for lock-less and concurrent reads/writes to the buffer by separating "ownership" of the internal variables such that there is at most one writer per variable. More specifically, each staging buffer is implemented as a circular queue with two variables marking the position where data can be added (*producerPos*) and another where data can be read from (*consumerPos*). Only the logging thread (the producer) can modify the producerPos, and only the NanoLog background thread (the consumer) can modify the consumerPos. However, both are free to read from either variable.

To understand how this design can allow for safe and concurrent reads/writes to the buffer without locks, consider the operation of a producer/logging thread in Figure 7.1(a-c). To add data to the queue, the producer thread checks that there is enough free space by reading both the producerPos and consumerPos and subtracting them to find the size of the free space region (the non-shaded regions in the figures). If there's enough space, the producer then copies data into the producerPos position, and bumps the producerPos pointer. If there's not, then it returns with an error and repeatedly retries. To read data from the buffer, the background thread performs a similar operation; it queries the consumable space, reads the data immediately after consumerPos and bumps the consumerPos pointer.

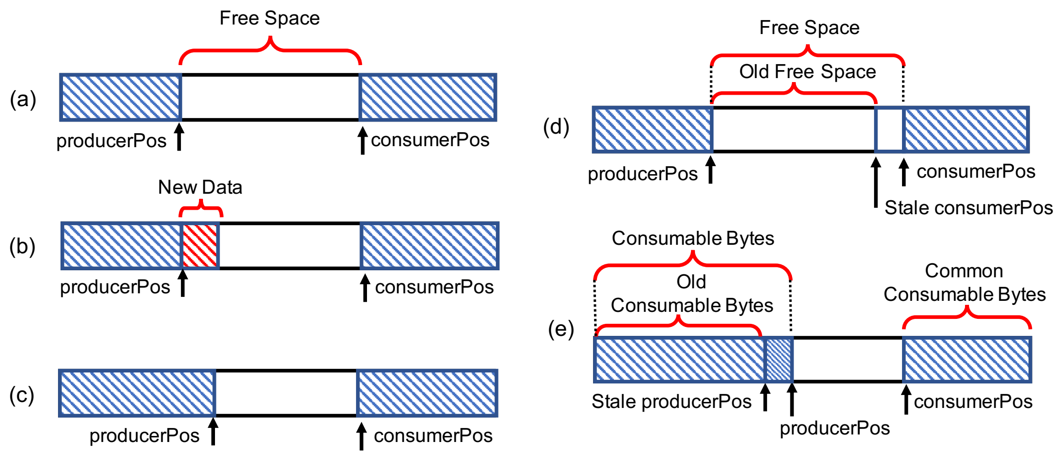


Figure 7.1: Portions (a-c) show the operations required to insert new data into a staging buffer and (d-e) shows what happens when stale values are used. (a) The staging buffer performs a “Free Space” calculation by taking the difference between the consumer position (consumerPos) and producer position (producerPos). (b) If there is enough space, the new data item is copied into the buffer at the producer position. (c) After the data is copied, the producer position is updated to after the new data added. (d) If a producer uses a stale value of consumerPos to calculate the free space (as could happen if the consumer concurrently consumed data and updated the pointer), then it will conservatively report less free space than the actual free space. (e) If a consumer uses a stale value of producerPos to calculate the consumable space, it will also conservatively report less space. In both cases, it’s safe to use the stale values to determine usable space, as they will both underestimate the true value (and not overwrite/overread data).

The important thing to notice is that the producer and consumer can operate concurrently in this design without causing corruption. Consider the producer in Figure 7.1(d). To calculate free space, it must query both the producerPos and consumerPos variables. The producer will always have an up-to-date value for producerPos, since it "owns" the variable, and only the consumerPos may be old, or outdated, as the consumer may have consumed space and written to consumerPos since the calculation started. However, it is safe for the producer to use the outdated consumerPos. Since "free space" can only increase with time (the consumerPos moves to the right in the figure), using an old value would only at worse underestimate the free space. This estimate, "old free space" in Figure 7.1(d), is a subset of the true "free space," so it's guaranteed to be truly free and safe to use. Thus, the producer can effectively ignore the fact that consumerPos may be updated by the consumer, since it is safe to use an older value. A similar argument can be made with the consumer and its usage of producerPos variable to determine the "consumable bytes" (Figure 7.1(e)).

Additionally, the memory operations are carefully ordered in the staging buffers to prevent inconsistent reads. In the case of the logging thread, it will place data into the buffer (with a store fence) before bumping the producerPos. This ordering of operations prevents the background thread from reading the log data before it is completely written, as the producerPos variable used in the space calculation would not be updated until the writes complete. Similarly, the background thread reads the log data (with a load fence) before modifying the consumerPos. This prevents the logging thread from overwriting data that has not yet been completely read by the background thread. And lastly, since only the logging thread can write to the buffer/producerPos and the background thread can only write to the consumerPos, there are no conflicting writes to the same variables[2].

One simplification made to the discussion above is the omission of an `endOfRecordedSpace` pointer. The pointer is owned by the logging thread/producer and is used to mark the last valid byte in the buffer. This pointer is needed when the producer runs out of space to store a complete log entry at the end of the buffer and needs to wrap around to the beginning. When this happens, the logging thread will mark the endOfRecordedSpace and reset producerPos to the beginning of the buffer to write the new log entry. The background thread can then use the pointer to detect when it has reached the end of valid data in the buffer and also needs to wrap around to continue reading data. This technique discards a bit of space at the end of the buffer, but it allows log statements to be written contiguously, which simplifies the log processing logic later. Again, care is taken to ensure that the endOfRecordedSpace pointer is written (with a store fence) before the producerPos

[2]Individual reads and writes to single-word values are assumed to be atomic.

is updated to ensure that the background thread will either see a stale value of producerPos and underestimate the space, or an up-to-date value of endOfRecordedSpace and wrap around in its space calculations.

Overall, the lock-less design of the staging buffers allows for concurrent and parallel access to the staging buffers. This increases the overall throughput of the system as both the logging and background threads can proceed in parallel (assuming multiple cores). Additionally, it lowers the log latency for the logging threads; they no longer have to wait for the background thread to complete its operations, and they do not need to acquire/release a lock for every operation.

7.5 Memoized Space Calculations

The current design induces excessive cache misses in highly active systems. Every read/write operation requires the logging and background threads to query both the consumerPos and producerPos variables to calculate the amount of space available. In a highly actively system, both variables would be constantly updated by the two threads as data is produced/consumed. Thus, a cache miss will be induced whenever one thread attempts reads the other's position. This cache miss can slow the operation by tens of nanoseconds; this is several times higher than the cost of logging. Even worse, it is possible for the logging and background threads to operate in lock step, causing cache misses for every operation as the threads repeatedly read and update the position variables. Fortunately, these cache misses can be mitigated through memoization.

The solution to this problem is to memoize the space calculations. More specifically, the producer and consumer will each save a private *space counter* indicating the amount of free/consumable space available in the buffer. They will then read and deduct from their private space counters for every operation until the variable is exhausted. Only when the space counter is exhausted would the producer or consumer take a cache miss and read both variables to update the private space counter. This strategy is equivalent to the producer saving the "Free Space" value from Figure 7.1a and deducting from it until the value reaches zero. Using this strategy reduces the number of cache misses and amortizes the cost of the space calculation.

This optimization is safe to perform for the same reason why utilizing a stale value for the producer/consumer positions is safe. We reasoned earlier that it is safe for the producer to read a stale value of the consumerPos for its free space calculation, because the free space can only grow with time. Saving a copy of the free space and deducting from it is equivalent to intentionally freezing the consumerPos to a specific stale value. The free space calculated from it is guaranteed to be less than or equal to the actual free space. Similarly, the amount of free space represented by the space counter is also guaranteed to be less than or equal to the actual free space, so using this value to calculate usable space is safe. An equivalent argument exists for the consumer and the producerPos pointer.

7.6 Mitigating False Sharing

The current design described so far can suffer from false sharing. False sharing occurs when multiple variables share the same cache line, and threads modifying independent variables within that cache line cause cache invalidations for the other threads. This is called "false sharing" because the threads may not actually be sharing the variables, but nonetheless cause cache invalidations. False sharing applies to the current design, because although the consumer and producer threads can now operate independently with their memoized space counters, the variables may still share the same cache line. As a result, whenever one thread updates their private space counter, it may cause a cache invalidation to occur for the other.

NanoLog mitigates this style of false sharing by utilizing cache line spacers. It places dummy arrays the size of a cache line in between variables that are exclusively written to by the producer and variables that are exclusively written to by the consumer. This effectively pushes the two sets of variables apart to separate cache lines and helps reduce false sharing. With this optimization, the logging and background threads write to their private variables without inducing superfluous cache invalidations for the other.

7.7 Polling and Avoiding Signals

Another optimization NanoLog utilizes is polling instead of signaling to notify the background thread of log data. One challenge the logging threads have is how to notify the background thread there's log data in one of the buffers. The traditional solution to this problem is to use C++ condition variables [2]. In this design, the logging threads would invoke `condition_variable::notify()` after every log message. This call notifies the background thread that there is log data and can wake up a potentially sleeping background thread. However, condition variables are too expensive for NanoLog. The notify operation takes on the order of 14 nanoseconds in my measurements, which is already double the target 7ns for logging. As a result, the logging threads simply place data into the staging buffers and do nothing. It relies on the background thread to round-robin between the

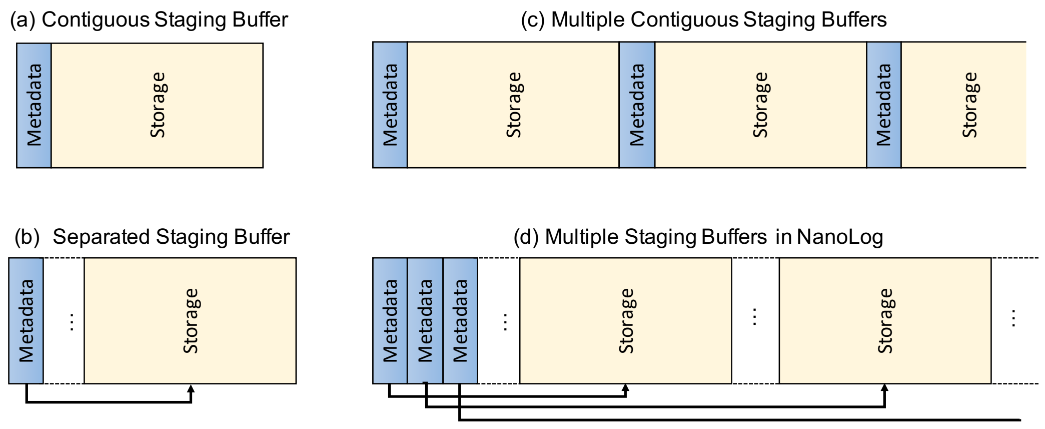

(a) Contiguous Staging Buffer
Metadata
Storage
(c) Multiple Contiguous Staging Buffers
Metadata
Storage
Metadata
Storage
Metadata
Storage
(b) Separated Staging Buffer
Metadata
…
Storage
(d) Multiple Staging Buffers in NanoLog
Metadata
Metadata
Metadata
…
Storage
…
Storage
…

Figure 7.2: The alternate designs of staging buffers (a-b) and their potential memory layout when multiple buffers are allocated (c-d). The blue sections represent staging buffer metadata, tan represents the log storage area, and white indicates unrelated memory areas. (a) Shows the design of a staging buffer where the metadata and bulk log storage are allocated adjacent to each other. (b) Shows the design where the metadata and bulk storage are allocated separately and the metadata has a pointer to the bulk storage. Portions (c-d) show the potential memory layout when multiple staging buffers of type (a-b) are allocated.

buffers and find the log data itself.

7.8 Coalesced Metadata

A final avenue for improvement is reducing the cache misses for the background thread. The current design of NanoLog requires the background thread to round-robin between the staging buffers and read their *metadata* (such as the position pointers or memoized space variables) to determine if there's log data to consume. Each metadata read can potentially incur a cache miss. However, through careful arrangement of the staging buffer's metadata, the cache misses can be mitigated.

The key to mitigating the cache misses is to store the metadata for multiple staging buffers in a contiguous chunk of memory[3]. When I initially designed the staging buffers, I had allocated the metadata to be contiguous with the log storage area for each staging buffer as in Figure 7.2a. This design meant that the metadata for each consecutively allocated staging buffer would be separated by at least the size of the log storage area (Figure 7.2c). This in turn resulted in slower performance

[3]The "metadata" in this case includes the producerPos/consumerPos pointers, the memoized space counters, and the cache line spacers that separate the consumer and producer variables.

for the background thread as the processor's prefetcher and speculative execution engine would not properly predict the next metadata to fetch. The optimization I came up with was to allocate the metadata separate from the staging buffer's log storage area and store the metadata from multiple staging buffers in a contiguous chunk of memory (Figure 7.2b,d). This allowed the background thread to poll the staging buffers more efficiently as the processor was able to execute speculatively and prefetch the next metadata. Overall, this reduced the cost to poll each buffer by about a cache miss.

7.9 Summary

The staging buffers employ a series of techniques to enable low-latency and high throughput operation. They are lock-free and allocated per thread to allow for concurrent logging and background thread operations. Shared variables are separated onto separate cache lines to avoid false sharing, and space counters are utilized to minimize references to shared variables. Finally, the staging buffers' metadata is allocated contiguously to mitigate cache misses for the background thread's polling operations.

Overall, these optimizations allow the staging buffers to operate in the range of single digit nanoseconds vs. the micro- to milliseconds incurred by a centralized, monitor-style queue.

Chapter 8

Compression

When I designed NanoLog, I knew that reducing I/O would play a large part in enabling nanosecond scale logging. After formatting, I/O is the second most expensive operation in a logging system. When I started the project, 125MB/s disks and gigabit networks were common, so outputting a typical 100-byte message would consume at least 800 nanoseconds of bandwidth[1]. Thus, NanoLog employs two strategies to deal with the I/O bottleneck. The first is that it utilizes a dictionary to deduplicate static, non-changing information in the log file, and the second is applying variable length encoding as a form of compression on the data remaining in the log file. The former transformation is described in detail in Chapter 4 and leaves the log file with only binary values to represent log statements. The second warrants more discussion and will be the main focus of this chapter.

In this chapter, I will describe the structure of the NanoLog log file to motivate the compression algorithm, explain why traditional dictionary-based compression schemes are inappropriate for NanoLog, and explain why variable length integer encodings are more performant. Finally, I will describe NanoLog's compression algorithm and conclude with some benchmarks and limitations.

8.1 Background

NanoLog uses a variable-length integer representation in the log file as the primary form of compression. This type of compression is most appropriate for the types of data that typically appear in a NanoLog log file and offers the best performance for modern I/O devices when compared to traditional, dictionary-based compression schemes.

[1]This is the best-case assuming large, batched I/O.

8.1.1 The Structure of Log Data

To understand the motivation behind NanoLog's compression strategy, we first have to understand the types of of data that NanoLog encounters.

The predominant data type in the NanoLog system is the integer type. Recall from Chapter 5 that the NanoLog log file primarily consists of log messages, which contain a 64-bit timestamp delta, a 32-bit identifier, and the message's dynamic arguments. This means that a log statement with no arguments is entirely represented by just two integers. Furthermore, integers are also the most common dynamic argument type. This claim is based on both personal observation and a survey of several popular, open-source projects that use printf-like statements (see Section 8.3.1). These two facts make integers the most common data type found in NanoLog log files.

Furthermore, the integers encountered by NanoLog tend to be small relative to their type defined container size. For example, NanoLog allocates 32-bits for the message identifiers, but most systems have fewer than 16,000 unique log messages. This means the identifiers could be represented in as few as 1-2 bytes, and a similar argument can be made about the 8-byte timestamp deltas between log messages. To be explicit, the NanoLog system allocates 8 bytes to store the absolute worst case time, but deltas less than 15ms can be represented in 3 or fewer bytes. Additionally, my experience informs me that users tend to log integers significantly smaller than their container size. Some popular examples include logging 64-bit metrics which have values in the billions, logging 32-bit thread ids that may have values in the hundreds, and printing out 32-bit enumerations which may only have values in the tens. Overall, the NanoLog log file tends to be comprised of many integers with small values.

8.1.2 Picking a Compression Scheme

When thinking of compression, one may be tempted to use traditional dictionary-based compression algorithms such as the LZ77 algorithm used in gzip[17]. However, they are not appropriate for the types of data and throughput produced by NanoLog. These algorithms specialize in deduplicating ASCII data by building a dictionary of repeating patterns. However, after passing through the NanoLog front-end, very little ASCII data remains in the NanoLog log file. Instead, the log file is mostly comprised of binary data in the form of integers such as timestamps, unique log identifiers, and user arguments[2]. Essentially, the NanoLong front-end already performs much of the work that

[2]The exception to this is if the user of NanoLog repeatedly logs strings.

Original (Decimal)	Original (Hex)	Tag (Binary)	Smallest Representation (Hex)	Byte Size
100	`0x0000000000000064`	`0001`	`0x64`	1.5
1	`0x0000000000000001`	`0001`	`0x01`	1.5
-1	`0xffffffffffffffff`	`1001`	`0x01`	1.5
50000	`0x000000000000c350`	`0010`	`0x0c35`	2.5
-50000	`0xffffffffffff3cb0`	`1010`	`0x0c35`	2.5
2ˆ62	`0x4000000000000000`	`1000`	`0x8000000000000000`	8.5
-2ˆ62	`0xc000000000000000`	`1000`	`0x8000000000000000`	8.5
50000	`0x0000000000000000000000000000c350`	`0010`	`0x0c35`	2.5
2ˆ127	`0x80000000000000000000000000000000`	`0000`	`0x80000000000000000000000000000000`	16.5
2ˆ64	`0x00000000000000010000000000000000`	`0000`	`0x00000000000000010000000000000000`	16.5
-2ˆ64	`0xffffffffffffffff0000000000000000`	`0000`	`0xffffffffffffffff0000000000000000L`	16.5

Table 8.1: Various endings of 64-bit and 128-bit numbers in the NanoLog compression scheme. The two leftmost columns represent the original integer to be compressed, the next two columns show encoded representation, and the last column shows the final byte size of the representation. Values above the horizontal divider represent 64-bit numbers and values below represent 128-bit numbers.

a dictionary-based compression scheme would perform.

Additionally, these dictionary-based algorithms are computationally too slow for high-throughput applications. They are optimized for maximal compression and will trade large amounts of computation time to save I/O time. On modern output devices, this trade-off rarely makes sense; it is often faster to output the raw data than it is to perform any sort of compression (see Section 8.3.2). As a result, modern high-throughput, low latency applications such as stream processing [48] and database queries [58] instead prefer a faster strategy that offers a better compute to compression trade-off: variable length encoding.

Variable length integer representations are a relatively simple way to reduce the size of integers with minimal computation. The base algorithm is to truncate the leading zero bytes to an integer and use a tag or marker bits to indicate the number of remaining bytes. For example, a 64-bit integer with the value 100 (or `0x0000000000000064` in hex) may be represented in as few as 1 byte in encodings such as *Little Endian Base-128* [10]. These algorithms also tend to be extremely fast as they only operate on a word at a time and do not require random memory access to perform lookups in a dictionary. This low cost and high performance makes variable length encoding an attractive choice for NanoLog.

8.2 NanoLog's Variable Length Encoding

At the time of NanoLog's development, I was unaware other variable length encoding implementations, so I developed a new one for NanoLog. The implementation is about as performant as the other variable length encoding schemes (Section 8.3.2) and does not appear to offer any clear advantages or disadvantages. Nonetheless, the scheme will be presented here for completeness.

The NanoLog system uses an encoding of a 4-bit tag followed by the number of bytes remaining after truncating the leading zeros. Table 8.1 shows some sample encodings. A tag with a decimal value from [1,8] indicates the number of remaining bytes encoded and a value between [9,15] indicates that [1,7] bytes have been encoded with a negation.

The negation representation exists to enable further compression on small negative numbers. For example, the value of -1 (`0xffffffffffffffff`) is normally incompressible since it contains no leading zero bytes. However, with a negation, the representation changes to `0x0000000000000001`, which is highly compressible due to the leading zeros. Note that the negation representation only exists for [1,7] bytes remaining, since an 8-bytes remaining negation would not save any additional space compared to the non-negated version.

Finally, a tag value of 0 indicates that a 128-bit, or 16-byte, integer has been encoded. This tag is used to support GNU GCC's arcane `__int128` extension[20]. Due to the limited number of tag bits, the system can only encode [1-8] and 16 bytes remaining. This means all representations that could ben encoded in [9,15] bytes would automatically be encoded as the full 16-bytes integer plus the 0.5 tag byte. The last two rows of Table 8.1 show examples of this encoding.

One peculiarity of this design is that nowhere is the width of the original integer type encoded. This information is particularly important to the NanoLog system as the post-processor needs to restore the correct argument type for use with `printf`. However, the omission is intentional. The reason why this type information has been omitted is because the front-end treats it as static information and instead encodes it into the dictionary for the post-processor. Thus, the type information does not need to be explicitly encoded here, and the post-processor can refer to the dictionary to retrieve it.

Lastly, when used in the NanoLog log file, all tag bits pertaining to a single log message are stored contiguously. This is done so that each pair of 4-bit tags forms a full byte. For example, if a log message contains two integer arguments *a* and *b*, the encoding would be 4-bits for tag *a* followed by 4-bits for tag *b* (which forms one full byte) followed by the remainders of *a* and *b*.

System Name	Static Characters	Integers	Floats	Strings	Others	Messages
Memcached	56.04	0.49	0.00	0.23	0.04	378
httpd	49.38	0.29	0.01	0.75	0.03	3711
linux	35.52	0.98	0.00	0.57	0.10	135119
Spark	43.32	n/a	n/a	n/a	n/a	2717
RAMCloud	46.65	1.08	0.07	0.47	0.02	1167

Table 8.2: The average number of static characters and dynamic variables in formatted log statements for five open source systems. These numbers were obtained by applying a set of heuristics to identify log statements in the source files and analyzing the embedded format strings; the numbers do not necessarily reflect runtime usage, and the heuristics may not have identified every log invocation. The "Messages" column counts the total number of log messages found in the source files. The argument type statistics are omitted for Spark since their logging system does not use format specifiers, and thus argument types could not be easily extracted. The static characters column represents the log message's format string with variable references (such as `"%4.2f"` or `"$variable"` in Spark) removed and represents the minimum number of characters that would be trivially saved by using NanoLog. The sources that generated this table are available on GitHub [69].

8.3 Evaluation

In this section, I will show that integers are the most common data type encountered by NanoLog, that Variable Length Encoding (VLE) strategies far outperform traditional dictionary-based algorithms for compressing integers, and that VLE algorithms are the best choice for compressing NanoLog's binary log file on modern hardware.

8.3.1 Log Message Types

The decision to use variable integer representation in NanoLog is based on personal observations that the most common argument type logged is integers. To verify this hypothesis, I wrote scripts to analyze five open-source projects that use printf-like logging APIs and report the average occurrence of integers, strings, and float specifiers in the projects' log messages' format strings[69]. Table 8.2 shows the results for Memcached [15], The Apache HTTP Server Project [61], RAMCloud [45], Spark [63], and linux [34].

In Table 8.2, all but one of the projects have integers as the most commonly logged data type. The Apache HTTP Server Project stands out as the only project logging the most strings. I suspect the reason why Apache logs more strings is because it primarily processes human-readable and highly variable HTTP requests and serves files. Thus, the log statements in the project typically output the header and file contents to aid in debugging. This is not an issue in the other projects as they deal more with metrics or binary data.

The other surprising finding from the table is that strings are the second most common argument type. This may inform us to use a different compression scheme for NanoLog. However, on closer inspection of the log messages themselves, it appears most of these strings can be represented in an integer fashion. For example, linux commonly prints the module name associated with a log message or C enumerations which are static and unchanging. With a stronger front-end, one could further extract these static components and integrate them into the dictionary. Chapter 10 discusses these potential enhancements in more detail.

One limitation of this survey is that it weighs all log statements in the applications' sources equally. It may be the case that certain log messages are executed more often than others at runtime (and others may never execute). This would result in a log message distribution that differs from what is shown in the table. Thus, one should only use this survey as a indicator of what *may* appear in a log file and not a representative of the ground truth at runtime.

Nonetheless, this survey provides us with the hint that applications mostly log integer values. The only exceptions are applications that deal with human-readable data such as httpd.

8.3.2 Variable Length Encoding Performance

In this section, I evaluate the effectiveness of variable length encoding (VLE) as a compression algorithm for NanoLog by comparing the performance of traditional dictionary-based compression vs. VLE for integer data types.

Table 8.3 shows the throughput and compression efficiency of various compression algorithms on an input of 1 million 64-bit integers that are uniformly distributed on bit-length. The left cluster of columns shows the throughput in MB/s and the right cluster shows the compression efficiency in the form of bytes/integer. Throughput is measured as the ingest rate for uncompressed data.

In this benchmark, the variable length encoding (VLE) algorithms outperform traditional algorithms by up to 3 orders of magnitude; most VLE algorithms can encode at a rate of 1 to 5GB/s for integers while gzip remains under 0.1GB/s. Additionally, the VLE algorithms maintain the smallest output sizes. Snappy closes the throughput gap at 3-8x slower than the compression speed of VLE algorithms, but that speed comes at a cost of a 30%-100% larger output size. Overall, the VLE algorithms outperform traditional dictionary based strategies for integer data types, making them viable for NanoLog's needs.

	Throughput (MB/s)				**Output Size (Bytes/Integer)**			
Algorithm	0-8 bits	0-16 bits	0-24 bits	0-64 bits	0-8 bits	0-16 bits	0-24 bits	0-64 bits
PrefixVarint	1362.78	839.25	691.32	571.97	1.12	1.69	2.25	5.07
LEB128	3345.44	1515.94	1146.37	746.12	1.12	1.69	2.25	5.08
leSQLite2	5132.32	1638.61	1172.22	853.66	1.07	1.66	2.31	5.25
leSQLite	5698.86	1614.49	1231.49	817.84	1.06	1.65	2.44	5.29
NanoLog	4671.51	1633.32	1279.63	940.74	1.50	2.00	2.50	5.00
gzip-1	95.45	68.14	57.12	35.78	1.62	2.46	3.16	5.76
gzip-6	22.33	16.15	13.94	14.04	1.39	2.27	2.95	5.57
gzip-9	1.57	1.32	1.28	1.87	1.27	2.17	2.86	5.53
snappy	769.40	475.48	387.35	303.02	2.24	3.29	4.17	6.57

Table 8.3: The encoding throughput of several variable length encoding algorithms taken from GitHub [31] and two dictionary-based compression libraries [17, 23]. The input is one million 64-bit numbers that are uniformly distributed in bit length (i.e. the position of the most significant bit is uniformly distributed). Each column limits the maximum value of each integer such the position of the most significant bit does not exceed 8, 16, 24, or 64. The left cluster of columns report the compression throughput, or the rate at which uncompressed data is ingested. The right cluster reports the compression efficiency, or the average number of bytes required to represent each integer in the dataset. The horizontal divider separates the variable integer representation algorithms (top) from the dictionary-based ones (bottom). The number after gzip indicates the compression level, where 1 is fastest, 9 is most compressed, and 6 is default.

Comparing NanoLog's algorithm to the other VLE's, NanoLog is not as performant nor as efficient for ultra small 0-8 bit numbers, but NanoLog closes the gap as the integer bit lengths increase. At 0-64 bits, NanoLog is the most performant and efficient compression algorithm. However at 0-8 bits, NanoLog's algorithm suffers as it requires an additional 4-bits to store the tag information. The main conclusion for this section is that one should use VLE algorithms for NanoLog's binary data and not necessarily that NanoLog has the best algorithm. If I were to design the system again, I would probably choose to use the leSQLite algorithm as it performs the best for smaller integers.

8.3.3 End-To-End Compression Throughput

NanoLog's compression mechanism is not very sophisticated in comparison to alternatives such as gzip [17] and Google snappy [23][3]. However, in this section I show that for compressing NanoLog's binary log data, NanoLog's approach provides a better overall balance between compression efficiency and execution time.

Figure 8.1 compares NanoLog, gzip, and snappy using 93 test cases. Each test case mimics a NanoLog log file with varying argument types, lengths, and entropy chosen to cover a range of log messages and to show the best and worst of each algorithm. For each test case and compression algorithm combination, I measured the overall logging throughput at a given I/O bandwidth. Here, the overall throughput is determined by the lesser of the compression throughput (i.e. CPU throughput) and I/O throughput (i.e. time to output the compressed data). Since the NanoLog background thread performs compression in parallel with disk writes, the slower of the two operations ultimately becomes the bottleneck at runtime. I then counted the number of test cases where an algorithm produced highest throughput of all algorithms at a given I/O bandwidth and graphed the results in Figure 8.1.

In Figure 8.1a, we see that aggressive compression only makes sense in low bandwidth situations; gzip,9 produces the best compression, but it uses so much CPU time that it only makes sense for devices with I/O bandwidth less than 5MB/s. As I/O bandwidth increases, gzip's CPU time quickly becomes the bottleneck for throughput, and compression algorithms that don't compress as much but operate more quickly become more attractive.

At the other extreme, "memcpy" performs the best with extremely high bandwidth devices

[3]Snappy is an LZ77 inspired compression algorithm that claims "memcpy-like" compression speeds.

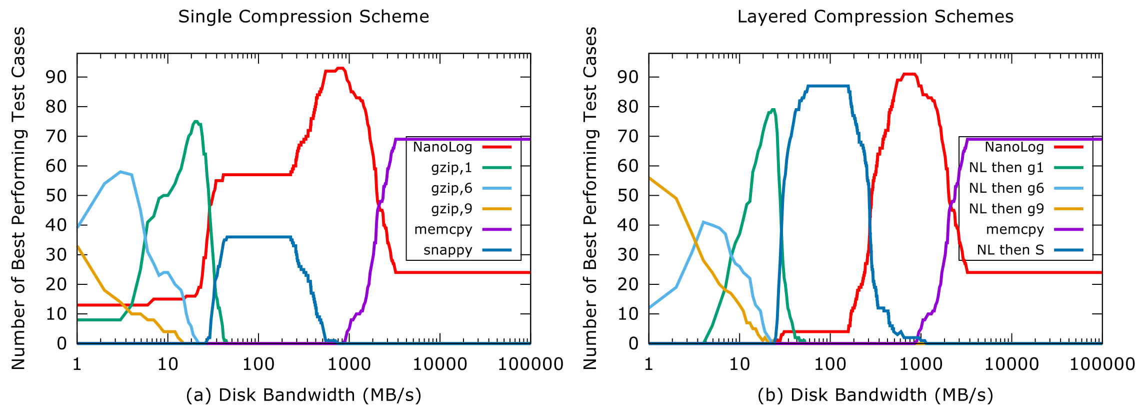


Figure 8.1: (a) shows the number of test cases (out of 93) for which each compression algorithm attained the highest throughput, and (b) shows the same except with layering where multiple compression algorithms were applied on top of each other. Here, "throughput" is defined as the minimum of an algorithm's compression throughput and I/O throughput (determined by output size and bandwidth). The numbers after the "gzip" labels indicate compression level, and "memcpy" represents "no compression." For figure (b), "NL" is NanoLog, "g" is gzip, and "S" is snappy. All combinations of the three algorithms were benchmarked, however only the highest performing algorithms are shown due to the limited space. The input test cases were 64MB chunks of binary NanoLog logs with arguments that varied in 4 dimensions: argument type (int/long/double/string), number of arguments, entropy, and value range. Strings had [10, 15, 20, 30, 45, 60, 100] characters and an entropy of "random", "zipfian" (θ=0.99), or "Top1000" (sentences generated using the top 1000 words from [42]). The numeric types had [1,2,3,4,6,10] arguments, an entropy of "random" or "sequential," and value ranges of "up to 2 bytes" and "at least half the type defined width." The full benchmark is available on GitHub [71].

(>2200MB/s). The “memcpy” line represents the minimal amount of computation that a compression algorithm has to do; it simply copies the data verbatim from one buffer to another. At extremely high bandwidth, this scheme tends to perform the best, as with large bandwidths, it’s often cheaper to simply output the raw data than to perform any sort of processing on it. Surprisingly, NanoLog is sometimes better than memcpy even for devices with extremely high I/O throughput. I suspect this is due to out-of-order execution on modern Intel processors[30], which can occasionally overlap NanoLog’s compression with loads and stores of the arguments; this makes NanoLog’s compression effectively free. However, I have not been able to verify this suspicion.

NanoLog provides the highest compression throughput for most test cases in the bandwidth range for modern disks and flash drives (30–2200 MB/s). The cases where NanoLog is not the best are those involving strings and doubles, which NanoLog does not compress; snappy is better for these cases. Overall, NanoLog’s compression scheme is the most efficient given the capability of current I/O devices and the types of data expected in a NanoLog log file.

As an academic curiosity, I also layered the compression algorithms and found that NanoLog enhanced the performance of dictionary-based compression schemes. Here, “layering” means piping the compression algorithms where the output of one is passed as input in the other. Figure 8.1b, shows the top performers of layering gzip, snappy, and NanoLog in all valid combinations[4]. The combination of NanoLog then gzip or snappy produced throughputs greater than either gzip or snappy alone in the low bandwidth region (<30MB/s). The addition of snappy ontop of NanoLog overtook NanoLog alone in the 30–300MB/s region, but NanoLog alone still dominated the 300MB/s+ range. Combinations of gzip and snappy did not provide a more performant algorithm. Overall, NanoLog is still extremely performant for high bandwidth devices, and layering snappy ontop of NanoLog may provide better performance for modern devices.

Lastly, while I believe NanoLog’s compression is superior for modern devices, the right compression algorithm to use ultimately depends on the types of information logged by the application and the I/O constraints of the operating environment. I therefore encourage those who are interested to download the benchmark and determine the right compression algorithm for themselves at `https://github.com/syang0/NanoLogCompression`.

[4]To be explicit, the combinations are gzip, gzip $\rightarrow$ snappy, snappy, snappy $\rightarrow$ gzip, NanoLog, NanoLog $\rightarrow$ snappy, NanoLog $\rightarrow$ gzip, NanoLog $\rightarrow$ snappy $\rightarrow$ gzip, NanoLog $\rightarrow$ gzip $\rightarrow$ snappy

8.4 Limitations and Extensions

The obvious limitations of using variable length encoding as a form of compression are (a) it only works with integers and (b) it only performs well for small integers. In fact, if an integer is full width (i.e. uses all bytes allowed by its type), then these variable length encoding schemes will typically *add* additional bytes per integer. This style of compression also does not operate on string and float types, which means if the user logs many strings, then traditional dictionary-based algorithms may be more viable than VLE.

Lastly, while traditional dictionary-based algorithms don't work well for the binary logs shown in the evaluation, they can still be useful in certain situations. For example, the dictionary created by the NanoLog front-end contains a lot of ASCII that can be compressed using traditional means; the format strings contain 45 characters on average (Table 9.2) and the source filenames removed by the front-end could contain repeated directory paths. However, the compression of the dictionary was not explored further, as it is considered an amortized cost.

8.5 Conclusion

Overall, I believe that variable length integer encodings are more effective form of compression than traditional dictionary-based algorithms for NanoLog's binary log data in the common case. However, depending on the bandwidth/compute limitations of the device and the types of data the user logs, other compression schemes may still be viable.

Chapter 9

Evaluation

In this chapter, I evaluate the performance of the NanoLog system for C++ applications. The library was written by me and is publicly available on GitHub [70]. In the library, the preprocessor/combiner component comprises about 1500 lines of Python, the C++17 front-end comprises about 1000 lines, and the NanoLog runtime/post-processor consists of about 6000 lines of C++ code.

I evaluated the system to answer the following questions:

- How does NanoLog's throughput and latency compare to other modern logging systems?
- How does NanoLog perform when embedded in a real, open-source application?
- What is the performance of the NanoLog post-processor?
- What performance gains can be had by directly querying the compressed, binary log file?
- What are NanoLog's primary bottlenecks?
- How does NanoLog's staging buffer size affect tail latency performance?

Additional experiments embedded in the other chapters (and not replicated here) include:

- How does NanoLog's compression scheme perform relative to other algorithms (Chapter 8)?

All experiments were conducted on quad-core machines with SATA SSDs that had a measured throughput of about 250MB/s for large writes (Table 9.1).

CPU	Xeon X3470 (4x2.93 GHz cores)
RAM	24 GB DDR3 at 800 MHz
Flash	2x Samsung 850 PRO (250GB) SSDs
OS	Debian 8.3 with Linux kernel 3.16.7
OS for ETW	Windows 10 Pro 1709, Build 16299.192

Table 9.1: The server configuration used for benchmarking.

9.1 System Comparison

To compare the performance of NanoLog with other systems, I ran microbenchmarks with six log messages (shown in Table 9.3) selected from an open-source datacenter storage system [45].

9.1.1 Test Setup

I chose to benchmark NanoLog against Log4j2 [62], spdlog [37], glog [21], Boost log [1], and Event Tracing for Windows (ETW) [47]. I chose Log4j2 for its popularity in industry; I configured it for low latency and high throughput by using asynchronous loggers and including the LMAX Disruptor library [35]. spdlog was chosen because it was the first result in an Internet search for "Fast C++ Logger"; I configured spdlog with a buffer size of 8192 entries (or 832KB). I chose glog because it is used by Google and configured it to buffer up to 30 seconds of logs. I chose Boost logging because of the popularity of Boost libraries in the C++ community; I configured Boost to use asynchronous sinks. I chose ETW because of its similarity to NanoLog; when used with Windows Software Trace PreProcessor [28], the log statements are rewritten to record only variable binary data at runtime. I configured ETW with the default buffer size of 64 KB; increasing it to 1 MB did not improve its steady-state performance.

I configured each system to output similar metadata information with each log message; they prepend a date/time, code location, severity, and thread id to each log message as shown in Figure 9.1. However, there are implementation differences in each system. In the time field, NanoLog and spdlog computed fractional seconds with 9 digits of precision (nanoseconds) vs. 6 for Boost-/glog and 3 for Log4j2 and ETW. In addition, Log4j2's code location information (ex. "TableManager.cc:1031") was manually encoded due to inefficiencies in its code location mechanism [64]. The other systems use the GNU C++ preprocessor macros "`__LINE__`" and "`__FILE__`" to encode the code location information.

```
NANO_LOG(NOTICE, "Creating table '%s' with id %d", name, tableId);

2017/3/18 21:35:16.554575617 TableManager.cc:1031 NOTICE[4]: Creating table
'orders' with id 11
```

Figure 9.1: A typical logging statement (top) and the resulting output in the log file (bottom). "NOTICE" is a log severity level and "[4]" is a thread identifier.

System Name	Static Chars	Integers	Floats	Strings	Others	Logs
Memcached	56.04	0.49	0.00	0.23	0.04	378
httpd	49.38	0.29	0.01	0.75	0.03	3711
linux	35.52	0.98	0.00	0.57	0.10	135119
Spark	43.32	n/a	n/a	n/a	n/a	2717
RAMCloud	46.65	1.08	0.07	0.47	0.02	1167

Table 9.2: Shows the average number of static characters (Static Chars) and dynamic variables in formatted log statements for five open source systems. These numbers were obtained by applying a set of heuristics to identify log statements in the source files and analyzing the embedded format strings; the numbers do not necessarily reflect runtime usage and may not include every log invocation. The "Logs" column counts the total number of log messages found. The dynamic counts are omitted for Spark since their logging system does not use format specifiers, and thus argument types could not be easily extracted. The static characters column omits format specifiers and variable references (i.e. $variables in Spark), and represents the number of characters that would be trivially saved by using NanoLog.

9.1.2 Choosing the Log Messages

The six log messages in Table 9.3 were selected to cover a spectrum of scenarios that seemed likely to occur in practice. They include messages where NanoLog offers the maximum performance benefit (with a completely static log message that can be removed at compile-time) and expected worst cases with floats and strings where NanoLog can perform no compression on the arguments. The selected log statements do not cover all possible combinations of log messages and dynamic arguments, but they should provide the reader with a feel for NanoLog's expected performance.

To ensure that the log messages chosen were representative of real world usage, I additionally statically analyzed log statements from five open source systems [15, 61, 34, 63, 45]; the results are in Table 9.2. The table shows that log messages have around 45 characters of static content on average and that integers are the most common dynamic type. Strings appear to be the second most common dynamic type, but upon manual inspection most strings logged are static. They typically contain pretty print error messages, enumerations, object variables, and other static/formatted types. This static information could in theory be also extracted by NanoLog and replaced with an identifier. However, since NanoLog does not currently perform this optimization, I will count these instances towards the "strings" datatype category. The log analyzer used to survey the log messages is a modified version of the NanoLog preprocessor and is publicly available on GitHub [69].

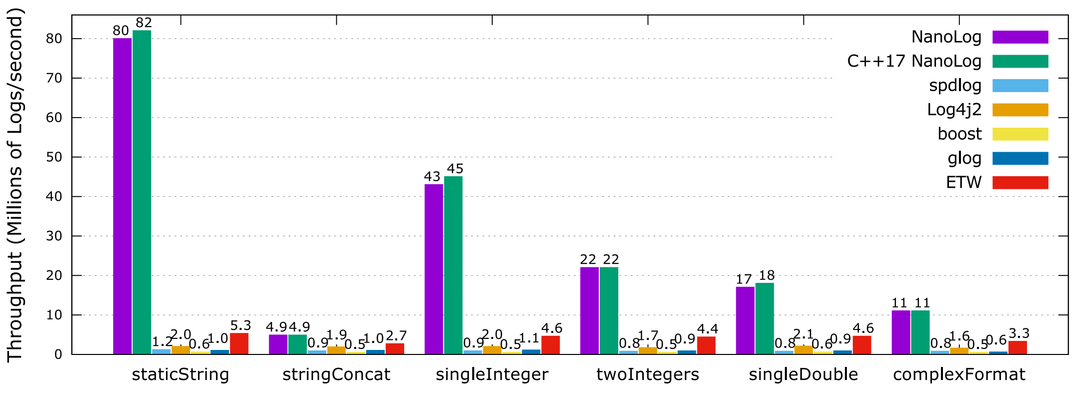


Figure 9.2: The maximum throughput attained by various logging systems when logging a single message repeatedly. Log4j2, Boost, spdlog, and Google glog logged the message 1 million times; ETW and NanoLog logged the message 8 and 100 million times respectively to generate a log file of comparable size. "NanoLog" in the figure refers to preprocessor NanoLog while "C++17 NanoLog" refers to C++17 NanoLog. The number of logging threads varied between 1 and 16 and the maximum throughput achieved is reported. All systems except Log4j2 include the time to flush the messages to disk in its throughput calculations (Log4j2 did not provide an API to flush the log without shutting down the logging service). The message labels on the x-axis are the same as those used in Table 9.3.

9.1.3 Throughput

Figure 9.2 shows the maximum throughput achieved by each logging system. NanoLog is faster than the other systems by 1.8x - 136.7x. The largest performance gap between NanoLog and the other systems occurs with *staticString*, and the smallest occurs with *stringConcat*.

NanoLog performs best when there is little dynamic information in the log message. This is reflected by *staticString*, a static message, in the throughput benchmark. Here, NanoLog only needs to output about 3-4 bytes per log message due to its compression and static log data extraction techniques. Other systems require over an order of magnitude more bytes to represent the messages (41-90 bytes). Even ETW, which uses a preprocessor to strip messages, requires at least 41 bytes in the static string case. NanoLog excels with static messages, reaching a throughput of 82 million log messages per second.

NanoLog performs the worst when there's a large amount of dynamic information. This is reflected in *stringConcat*, which logs a large 39 byte dynamic string. NanoLog performs no compression on string arguments and thus must log the entire string. This results in an output of 41-42 bytes per log message and drops throughput to about 4.9 million log messages per second. Even in this

ID	Example Output
staticString	Starting backup replica garbage collector thread
stringConcat	Opened session with coordinator at basic+udp:host=192.168.1.140,port=12246
singleInteger	Backup storage speeds (min): 181 MB/s read
twoIntegers	Buffer has consumed 1032024 bytes of extra storage, current allocation: 1016544 bytes
singleDouble	Using tombstone ratio balancer with ratio = 0.4
complexFormat	Initialized InfUdDriver buffers: 50000 receive buffers (97 MB), 50 transmit buffers (0 MB), took 26.2 ms

Table 9.3: Log messages used to generate Figure 9.2 and Table 9.4. The underlines indicate dynamic data generated at runtime. *staticString* is a completely static log message, *stringConcat* contains a large dynamic string, and other messages are a combination of integer and floating point types. Additionally, the logging systems were configured to output each message with the context "YY-MM-DD HH:MM:SS.ns Benchmark.cc:20 DEBUG[0]:" prepended to it.

scenario, NanoLog will consistently outperform its competitors, as it will always output fewer bytes than systems that materialize the full, human-readable log message with context.

Overall, NanoLog is faster than all other logging systems tested. This is primarily due to NanoLog consistently outputting fewer bytes per message and secondarily because NanoLog defers the formatting and sorting of log messages.

9.1.4 Latency

NanoLog lowers the logging thread's invocation latency by deferring the formatting of log messages. This effect can be seen in Table 9.4. NanoLog's *invocation latency*, or time between when the application executes the log function and when it returns, is 18-500x lower than other systems. In fact, NanoLog's 50/90/99/99.9th percentile latencies are all within tens of nanoseconds while the median latencies for the other systems *start* at hundreds of nanoseconds.

All of the other systems except ETW require the logging thread to either fully or partially materialize the human-readable log message before transferring control to the background thread, resulting in higher invocation latencies. NanoLog on the other hand, performs no formatting and simply pushes all arguments to the staging buffer. This means less computation and fewer bytes copied, resulting in a lower invocation latency.

Although ETW also produces binary log files like NanoLog, its latencies are still much higher than those of NanoLog. I am unsure why ETW is slower than NanoLog, but one hint is that even with the preprocessor, ETW log messages are larger than NanoLog (41 vs. 4 bytes for staticString).

ID	NanoLog				NanoLogCpp17				spdlog				ETW			
Percentiles	*50*	*90*	*99*	*99.9*	*50*	*90*	*99*	*99.9*	*50*	*90*	*99*	*99.9*	*50*	*90*	*99*	*99.9*
staticString	7	8	10	27	7	8	10	34	230	236	323	473	180	187	242	726
stringConcat	8	9	25	37	7	8	10	28	436	494	1579	1641	208	218	282	2954
singleInteger	7	8	8	29	7	8	10	34	353	358	408	824	189	195	237	720
twoIntegers	7	8	8	32	7	8	8	27	674	698	807	1335	200	207	237	761
singleDouble	8	8	9	35	7	7	10	34	607	637	685	1548	187	193	248	720
complexFormat	7	8	10	34	7	8	10	34	1234	1261	1425	3360	242	252	304	1070

ID	Log4j2				glog				Boost			
Percentiles	*50*	*90*	*99*	*99.9*	*50*	*90*	*99*	*99.9*	*50*	*90*	*99*	*99.9*
staticString	192	311	470	1868	1201	1229	3451	5231	1619	2338	3138	4413
stringConcat	230	1711	3110	6171	1235	1272	3469	5728	1833	2621	3387	5547
singleInteger	223	321	458	1869	1250	1268	3543	5458	1963	2775	3396	7040
twoIntegers	160	297	550	1992	1369	1420	3554	5737	2255	3167	3932	7775
singleDouble	157	252	358	1494	2077	2135	4329	6995	2830	3479	3885	7176
complexFormat	146	233	346	1500	2570	2722	5167	8589	4175	4621	5189	9637

Table 9.4: Unloaded tail latencies of NanoLog and other popular logging frameworks. This experiment was performed by a single application thread repeatedly logging a single log message back to back 100,000 times with a 600 nanosecond delay between each invocation to ensure that I/O is not a bottleneck. Latency is measured by the application as the time between when the application invokes the log function (e.x. NANO_LOG) to when the function returns. Each row shows the results for a different log message (taken from Table 9.3), and each column represents the 50th/90th/99th/99.9th percentile latency in nanoseconds. "NanoLog" represents Preprocessor NanoLog while "NanoLogCpp17" represents C++17 NanoLog. The bottom table is an extension of the top table and represents the same experiment.

ETW also emits extra log information such as process ids and does not use the efficient compression mechanism of NanoLog to reduce its output. There may be more differences between the two systems, however I was unable to measure further on ETW further as it is closed-source.

Furthermore, NanoLog's performance is extremely consistent in this microbenchmark. Although it may not be immediately clear, the 600 nanosecond delay between each log message (see caption in Table 9.4) was added primarily for the benefit of the traditional, ASCII-based logging systems. These systems materialized log messages that required around 100 bytes to represent, so a delay of 600 nanoseconds was chosen to account for 150 bytes of I/O on our 250MB/s disks. NanoLog does not need such a delay to maintain its low latency; a later benchmark (Section 9.6) shows that even with the delay removed, NanoLog still maintains a 99.9th percentile within tens of nanoseconds.

One caveat with these results is that they may represent an unrealistic, best-case performance for all logging systems. The application in this microbenchmark is only performing logging and is not performing any other operation that can interfere with the logging latency, such as contending for I/O and compute resources or polluting the instruction/data caches. This means that the performance reported here may be overoptimistic, and the invocation latencies may actually be higher when integrated in a real application. However since this caveat applies to all logging systems benchmarked, this microbenchmark should give readers a sense of NanoLog's relative performance (1-2 orders of magnitude faster than other systems). Section 9.4 evaluates a subset of the logging systems (NanoLog and spdlog) in a more realistic environment.

Overall, NanoLog's unloaded invocation latency is extremely low; it is 1-2 orders of magnitude lower than other logging systems.

9.2 Decompression

Since the NanoLog runtime outputs the log in a binary format, it is also important to understand the performance implications of transforming it back into a human readable log format.

The post-processor can decompress/format messages at a peak of about 0.5M log messages/second (Figure 9.3). Traditional logging systems such as Log4j2 can achieve a higher throughput of over 2M log messages/second at runtime, because they utilize multiple threads to perform formatting. NanoLog's post-processor is currently single threaded, so its performance is limited to that of a single thread. I made the assumption that the decompression operation is fairly uncommon, so I kept the post-processor single threaded for simplicity. However, NanoLog's post-processor could be modified to use multiple threads to achieve higher throughput.

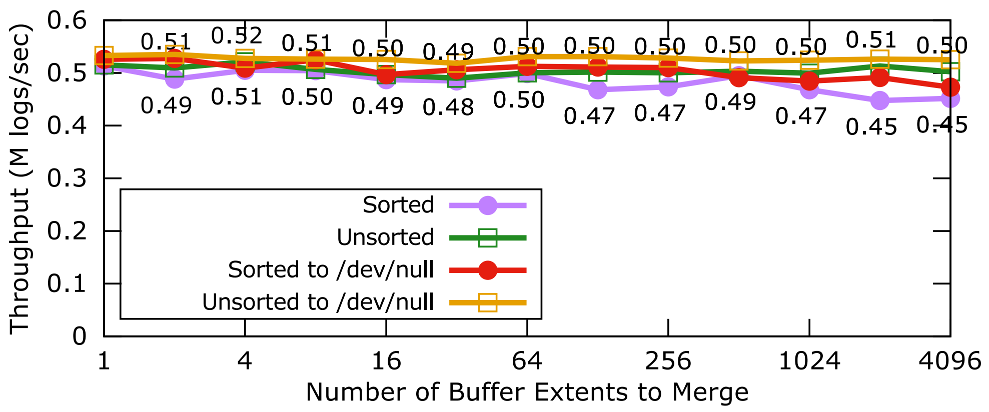


Figure 9.3: Impact on NanoLog's post-processor performance as the number of buffer extents that need to be merged increases. The post-processor was tested in four configurations: "Sorted" where the system outputs log messages in a chronological order to disk (this is the default/expected operating mode), "Unsorted" where the system outputs the log messages to disk without ordering the log messages, and "Sorted to /dev/null" and "Unsorted to /dev/null" which are the same as the above except they discard the log messages to /dev/null. The log files decompressed contained 2^{24} log messages (about 16M) and were formatted in the form of "2017-04-06 02:03:25.000472519 Benchmark.cc:65 NOTICE[0]: Simple log message with 0 parameters". The compacted log file was 49MB and the resulting decompressed log output was 1.5GB. The number of buffer extents (x-axis) was varied by increasing the number of concurrent logging threads at runtime.

The throughput of the post-processor could potentially drop if the post-processor is required to sort the log messages by time and there were many active logging threads when the log was created. The reason is that the log is divided into different buffer extents for each logging thread, and the post-processor must collate and sort the log messages from multiple extents into a single chronological ordering. This is done by performing a merge sort amongst the active buffer extents (two per thread) and using a heap to find the next log message to decompress.

The “Sorted” line in Figure 9.3 shows that sorting only has a minimal impact on decompression costs. Even with 4096 active logging threads, the post-processor’s decompression throughput only drops by about 10%. Increasing the logging threads beyond 4096 active logging threads may degrade performance further, however this setup was not tested due to practical limitations; modern applications would not utilize that many threads, and 4096 threads already far overwhelms our 4-core experimental setup. However, I suspect that the performance would continue to drop logarithmically with increasing logging threads, as the post-processor uses a heap to manage the active buffer extents (Note: An earlier publication of NanoLog [73] showed a linear degradation in performance with increasing buffer extents. This was because the post-processor used a linear sort rather than a heap at the time).

Sorting is only necessary if order matters during post-processing. For some applications, such as analytics, the order in which the log messages are processed may be unimportant. In these cases, sorting can be skipped. The “Unsorted” line in Figure 9.3 shows that decompression in these cases is unaffected by the number of runtime logging threads; the throughput remains steady at about 500,000 log messages/second.

Furthermore, Figure 9.3 shows that the post-processor is compute-bound. We see similar performance from the post-processor whether it persists the log messages to disk or discards them to “/dev/null”. This indicates that the bottleneck in the system is not from writing the log messages to disk, but rather from formatting/sorting. Digging a little deeper, I found that most of the decompression cost comes from formatting the log message. Each log message requires about two microseconds to process: about 1 microsecond is spent on formatting the time into a “YY-MM-DD HH:MM:SS.ns” format and another is spent formatting the log context information (as shown in Figure 9.1). Very little time is spent interpreting the data in the log statement (Section 9.3 quantifies this cost further), and only about 10% of the time is spent on sorting at 4096 threads. This suggests that the post-processor could easily utilize multiple formatting threads for additional speedups.

For completeness, the same evaluation was performed with all the log messages from Table 9.3. The results, shown in Table 9.5, show that the cost to decompress each message depends on how

	Sorted							**Unsorted**						
	Number of Buffer Extents to Merge							Number of Buffer Extents to Merge						
ID	1	4	16	64	256	1024	4096	1	4	16	64	256	1024	4096
staticString	0.48	0.49	0.48	0.45	0.47	0.45	0.42	0.50	0.51	0.50	0.51	0.50	0.50	0.50
stringConcat	0.47	0.46	0.46	0.44	0.44	0.45	0.41	0.48	0.47	0.46	0.48	0.47	0.47	0.47
singleInteger	0.47	0.45	0.46	0.46	0.45	0.45	0.42	0.49	0.49	0.48	0.48	0.49	0.47	0.48
twoIntegers	0.45	0.43	0.43	0.41	0.42	0.41	0.40	0.45	0.44	0.45	0.44	0.44	0.44	0.44
singleDouble	0.44	0.42	0.40	0.43	0.43	0.40	0.38	0.44	0.45	0.45	0.43	0.44	0.43	0.43
complexFormat	0.34	0.31	0.33	0.33	0.33	0.32	0.31	0.34	0.34	0.34	0.33	0.34	0.34	0.34

Table 9.5: The decompression performance of the post-processor with a varying number of overlapping buffer extents and log messages. The log messages are shown in the first column; they're taken from Table 9.3 and each datum shows the decompression throughput in millions of log messages per second. The log messages were decompressed to disk, each log file contained about 2^{23} (about 8M) log messages, and the messages were formatted with the header "2017-04-06 02:03:25.000472519 Benchmark.cc:65 NOTICE[0]: " followed by the log message as specified in Table 9.3. The "Sorted" columns indicate that the post-processor ordered the log messages by time before output whereas the "Unsorted" columns indicate that the system output the log messages without ordering them first.

much formatting and sorting is required. The message with the least formatting required, *staticString*, can achieve a throughput up to 0.51 million log messages per second and the log message with the most, *complexFormat*, achieves 0.34 million log messages per second. As with the previous results, the throughput is nearly constant up to around 1024 threads when sorting is required, and drops off with more threads as the sorting costs increase. Otherwise, the performance remains constant with no sorting. The results shown in Table 9.5 are consistent with the previous microbenchmark in Figure 9.3.

Overall, NanoLog can decompress log messages at rate of about 500,000 messages/second in applications with fewer than 1024 active logging threads, or in systems where log messages do not need to be processed in chronological order. The post-processor can potentially be improved via multi-threading support, however that is not explored here.

9.3 Aggregation Performance

NanoLog's compact, binary log output promises more efficient log aggregation/analytics than its full, uncompressed counterparts. To demonstrate this, I implemented a simple min/mean/max aggregation in four systems, NanoLog, C++, Awk, and Python. Conceptually, they all perform the same task; they search for the target log message "`Hello World # %d`", read the value of the "`%d`" integer, and perform calculations on the integer. They track of the minimum and maximum

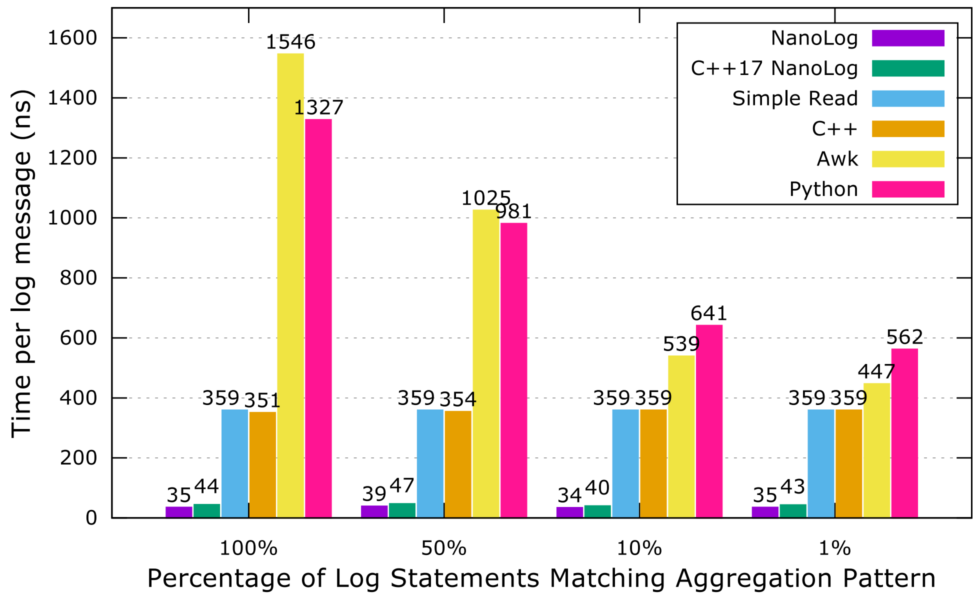


Figure 9.4: The average processing time per log message to perform a min/mean/max aggregation over 100 million log messages using various systems. The x-axis varies the percentage of log messages matching the target aggregation pattern "`Hello World # %d`" and the rest "`UnrelatedLog #%d`". The NanoLog system operated on a compacted file (∼747MB) and the remaining systems operated on the full, uncompressed log (∼7.6GB). The C++ application searched for the "`Hello World # `" prefix and utilized `atoi()` on the next word to parse the integer. The Awk and Python applications used a simple regular expression matching the prefix: "`.*Hello World # (\d+)`". "Simple Read" reads the entire log file and discards the contents. The file system cache was flushed before each run. "NanoLog" refers to the "Pure Preprocessor" version of NanoLog.

values encountered, as well as calculate the arithmetic mean by summing all values and dividing by the number of values. The difference between the systems is that the C++, Awk, and Python implementations operate on the full, uncompressed version of the log while the NanoLog post-processor operates directly on the output from the NanoLog runtime. The NanoLog post-processor also did not time-order the log statements or format them before processing.

Figure 9.4 shows the execution time for this aggregation over 100M log messages. NanoLog is nearly an order of magnitude faster than the other systems, taking on average 35-44 nanoseconds per log message to process the compact log file vs. 350+ ns for the other systems. The primary reason for NanoLog's low execution time is disk bandwidth. The compact log file only amounted to about 747MB vs. 7.6GB for the uncompressed log file. In other words, the aggregation was disk bandwidth limited and NanoLog used the least amount of disk IO. I verified this assumption with a simple C++ application that performs no aggregation and simply reads the file ("Simple Read" in the figure); its execution time lines up with the "C++" aggregator at around 359 nanoseconds per log statement.

I also varied how often the target log message "`Hello World # %d`" occurred in the log file to see if it affects aggregation time. The compiled systems (NanoLog and C++) have a near constant cost for aggregating the log file while the interpreted systems (Awk and Python) have processing costs correlated to how often the target message occurred. More specifically, the more frequent the target message, the longer the execution time for Awk and Python. I suspect the reason is because the regular expression systems used by Awk and Python can quickly disqualify non-matching strings, but perform more expensive parsing when a match occurs. However, I did not investigate further.

Lastly, "pure" preprocessor NanoLog performs about 20% better than C++17 NanoLog in post-processing (35 ns vs. 45 ns per log message). This is due to the architectural differences between the two systems. In particular, C++17 NanoLog requires logic to interpret the log statements at post-processing time while preprocessor NanoLog directly compiles specialized code for each log statement into the post-processor application. This results in less work being performed at post-processing and allows the compiler to perform more aggressive optimizations, netting a 20% performance improvement.

Overall, the compactness of the NanoLog binary log file allows for fast aggregation relative to traditional log processing methods.

There are two shortcomings I'd like to address with this benchmark. First, the aggregation was

		No Logs	NanoLog	spdlog	RAMCloud
Throughput (kop/s)	Read	994 (100%)	809 (81%)	122 (12%)	67 (7%)
	Write	140 (100%)	137 (98%)	59 (42%)	32 (23%)
Read Latency (μs)	50%	5.19 (1.00x)	5.33 (1.03x)	8.21 (1.58x)	15.55 (3.00x)
	90%	5.56 (1.00x)	5.53 (0.99x)	8.71 (1.57x)	16.66 (3.00x)
	99%	6.15 (1.00x)	6.15 (1.00x)	9.60 (1.56x)	17.82 (2.90x)
Write Latency (μs)	50%	15.85 (1.00x)	16.33 (1.03x)	24.88 (1.57x)	45.53 (2.87x)
	90%	16.50 (1.00x)	17.08 (1.04x)	26.42 (1.60x)	47.50 (2.88x)
	99%	22.87 (1.00x)	23.74 (1.04x)	33.05 (1.45x)	59.17 (2.59x)

Table 9.6: The impact on RAMCloud [45] performance when more intensive instrumentation is enabled. The instrumentation adds about 11-33 log statements per read/write request with 1-3 integer log arguments each. “No Logs” represents the baseline with no logging enabled. “RAMCloud” uses the internal logger while “NanoLog” and “spdlog” supplant the internal logger with their own. The percentages next to Read/Write Latency represent tail latency percentiles, and all results were measured with RAMCloud’s internal benchmarks with 16 clients used in the throughput measurements. Throughput benchmarks were run for 10 seconds and latency benchmarks measured 2M operations. Each configuration was run 15 times and the best case is presented.

deliberately simple. It only required the post-processor to track four variables: a minimum, a maximum, a sum, and a counter[1]. The aggregation did not include more complex operations such as tracking dependencies between log messages or correlating events, which may incur additional costs. It was explicitly chosen to showcase the minuscule overheads of using NanoLog; a more complex aggregation may require so much compute that it hides the performance differences between NanoLog and the other systems. Second, the aggregation was performed without first sorting the log messages by time. The reason why sorting was not measured is because it is currently not implemented in the proof-of-concept aggregation API. Additional costs may be incurred if sorting was required, as hinted by the decompression benchmark (Section 9.2).

9.4 Integration Benchmark

While microbenchmarks are useful to quantify the best-case performance in a vacuum, I also attempted to gauge NanoLog’s real world performance by integrating it into an application. I integrated preprocessor NanoLog and spdlog into a well instrumented, open-source key value store,

[1]The sum and counter are divided to retrieve the arithmetic mean.

RAMCloud [45], and evaluated the logging systems' impact on performance using existing benchmarks in the RAMCloud repository. In keeping with the goal of increasing visibility, I enabled all verbose logging options and changed existing performance sampling statements in RAMCloud (normally compiled out) to always-on log statements. This added an additional 11-33 log statements per read/write request in the RAMCloud system. With this heavily instrumented system, I could answer the following questions: (1) how much of an improvement does NanoLog provide over other state-of-the-art systems in this scenario, (2) how does NanoLog perform in a real system compared to microbenchmarks and (3) how much does NanoLog slow down compilation and increase binary size?

Table 9.6 shows that, with NanoLog, the additional instrumentation introduces only a small performance penalty. Median read/write latencies increased only by about 3-4% relative to an uninstrumented system and write throughput decreased by 2%. Read throughput sees a larger degradation (about 19%)[2]. In contrast, the other logging systems incur such a high performance penalty that this level of instrumentation would probably be impractical in production: latencies increase by 1.6-3x, write throughput drops by more than half, and read throughput is reduced to roughly a tenth of the uninstrumented system (8-14x). These results show that NanoLog supports a higher level of instrumentation than other logging systems.

Using this benchmark, we can also estimate NanoLog's invocation latency when integrated in a low-latency system; i.e. in a non-microbenchmark setting. For RAMCloud's read operation, the critical path emits 8 log messages out of the 11 enabled. On average, each log message increased latency by (5.33-5.19)/8 = 17.5 ns. For RAMCloud's write operation, the critical path emits 27 log messages, suggesting an average latency cost of 17.7 ns. These numbers are higher than the median latency of 7-8 ns reported by the microbenchmarks in Table 9.4, but they are still reasonably fast. I suspect the slowdowns are caused by cache misses on the staging buffers and are potentially unavoidable. The calculated latencies for spdlog are closer to the microbenchmark's results (around 380ns). I suspect this is because the operation is so expensive that slowdowns due to cache misses are lost in the noise. However, I have not been able to measure one level deeper to confirm either of these suspicions.

Lastly, I compared the compilation time and binary size of RAMCloud with and without NanoLog. Without NanoLog, building RAMCloud takes 9.27 minutes and results in a binary size

[2]This additional degradation occurs because the RAMCloud dispatch thread is already bottlenecked for reads [45] and the benchmark made it worse by adding additional logging to that thread.

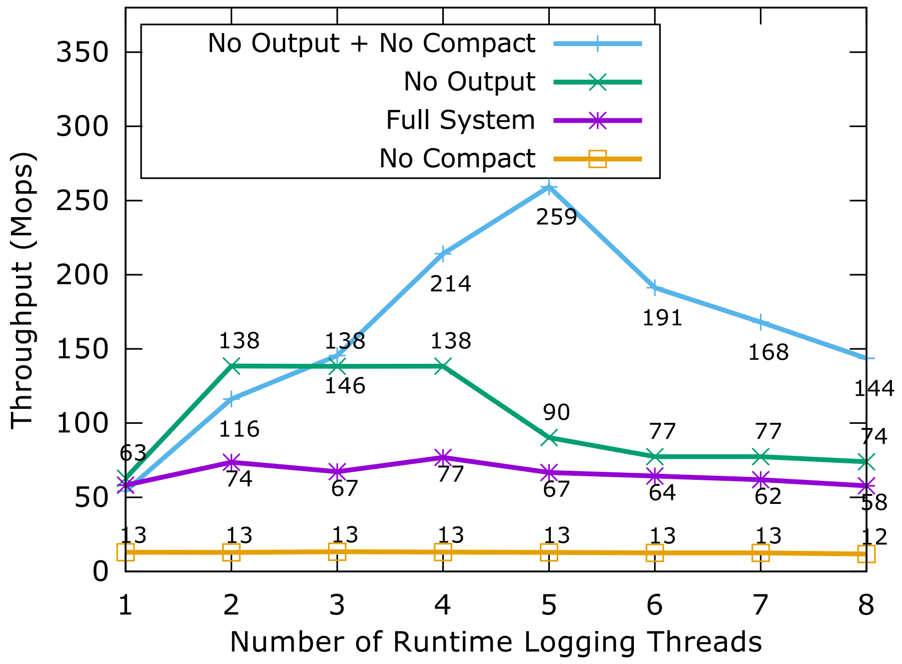


Figure 9.5: Runtime log message throughput achieved by the preprocessor NanoLog system as the number of logging threads is increased. For each point, 2^{27} (about 134M) static messages were logged. The *Full System* is Preprocessor NanoLog as described in this dissertation, *No Output* pipes the log output to /dev/null, *No Compact* omits compression in the NanoLog background thread and directly outputs the staging buffers' contents, and *No Output + No Compact* is a combination of the last two.

of 122 MB. With preprocessor NanoLog, the build time increased by 97 seconds (+17%), and the binary size increased by 19 MB (+16%). With C++17 NanoLog, the build time only increased by 9 seconds (+2%), and the binary size increased by 23 MB (+ 19%). Compilation times were measured by invoking "`make`" without any parameters, and the binary size is the sum of the coordinator executable, sever executable, and RAMCloud library. The dictionary of static log information amounted to 229KB for 922 log statements ($\sim$ 248B/message). The log message count differs from Table 9.2 because RAMCloud compiles out certain log messages depending on build parameters.

9.5 Throughput Bottlenecks

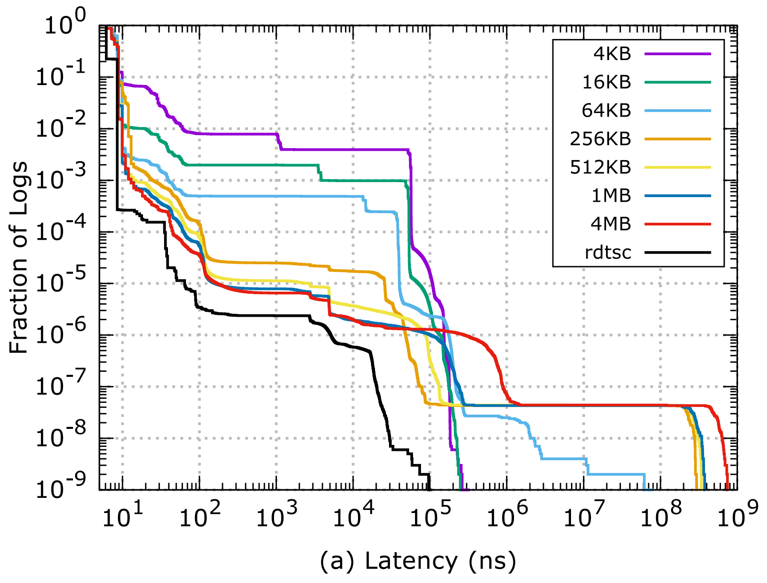
I was curious about how NanoLog's performance would scale with newer technologies, so I created a benchmark that would remove NanoLog's greatest bottleneck: I/O bandwidth. NanoLog's performance is limited by I/O bandwidth in two ways. First, the I/O bandwidth itself is a bottleneck for outputting raw log messages. Second, the compression that NanoLog performs in order to reduce the I/O cost can make NanoLog compute bound as I/O speeds improve. Figure 9.5 explores the limits of the system by removing these bottlenecks for preprocessor NanoLog.

Compression plays a large role in improving NanoLog's throughput, even on our testbed's SATAII flash devices (250MB/s). The "Full System" as described in the dissertation achieves a throughput of nearly 80 million operations per second while the "No Compact" system only achieves about 13 million operations per second. This is due to the 5x difference in I/O size; the full system outputs 3-4 bytes per message while the no compression system outputs about 16 bytes per message.

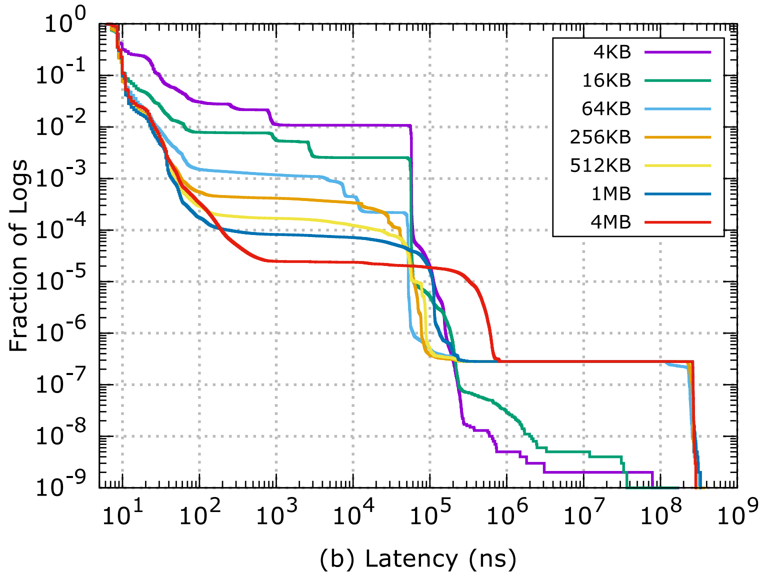
If I remove the I/O bottleneck altogether by redirecting the log file to `/dev/null`, NanoLog "No Output" achieves an even higher peak throughput of 138 million logs per second. At this point, the compression becomes the bottleneck of the system. Removing both compression and I/O allows the "No Output + No Compact" system to push upwards of 259 million operations per second.

Since the "Full System" throughput was achieved with a 250MB/s disk and the "No Output" has roughly twice the throughput, one might assume that compression would become the bottleneck with I/O devices twice as fast as ours (500MB/s). However, that would be incorrect. To sustain 138 million logs per second without compression, one would need an I/O device capable of 2.24GB/s (138M logs/sec x 16B).

Lastly, I suspect I was unable to measure the maximum processing potential of the NanoLog background thread in "No Output + No Compact." Our testbed only had 4 physical cores with 2 hyperthreads each; beyond 4-5, the logging threads start competing with the background thread for physical CPU resources, lowering throughput. Since the background thread is primarily performing memory copies in the "No Output + No Compact" scenario, I suspect the true maximum throughput would be a function of the message size (16B) divided by the machine's memcpy speed. However, no additional measurements have been made to confirm or deny this suspicion.



(a) Latency (ns)



(b) Latency (ns)

Figure 9.6: Complementary CDF showing the effect of staging buffer size on preprocessor NanoLog's tail latency when log messages are generated rapidly enough to bottleneck on I/O. The top figure uses the *staticString* message and the bottom figure uses *complexFormat*. Each curve represents 1 billion log messages logged back to back with no delay by one thread. A point (x, y) in the graph means that a fraction y of all messages had an invocation latency of at least x nanoseconds. The cliff that occurs near 10^5 nanoseconds for smaller buffers (4KB - 512KB) is due to the background thread intentionally sleeping between rounds of polling through the staging buffers. Finally, the "rdtsc" line in (a) is a microbenchmark that records 1 billion consecutive Intel Timestamp Counter [29] readings into a contiguous byte array.

9.6 Tail Latency and Staging Buffer Size

This section evaluates the extreme tail latency performance of NanoLog and how it changes with different sizes of the staging buffer. Under normal operating conditions, the staging buffer decouples the speed of the logging application from the speed of compression and I/O, and the logging application sees low latency for logging. However, if log messages are produced faster than the underlying I/O device can handle them, then the buffers back up and the logging thread will experience additional delays. Under these conditions, the size of the staging buffers impacts the frequency and length of the delays experienced by the logging application.

Figure 9.6 shows the distribution of latencies experienced by an application with a single thread that is generating log messages faster than they can be output. Each curve represents a different size for the staging buffer, and the two graphs represent two different log messages being repeatedly logged by a single thread. Each line outlines a shape where the left-most region is a sharp drop, followed by a plateau, followed by two sharp drops on the right at around 50µs and 100+ milliseconds. The regions where these transitions occur will be referred to as "*knees*" in the rest of this discussion.

Looking at Figure 9.6, we see that regardless of the staging buffer size, the NanoLog system achieves a 90th percentile latency (10^{-1}) under 10 nanoseconds. This range of low latencies occurs while the staging buffer still has free space to fill, allowing the logging thread to execute unimpeded at the highest performance. Once the staging buffer fills up, the logging thread is blocked and experiences additional delays.

Furthermore, we see that the larger the staging buffer size, the larger the fraction of log messages that can complete in under 10 ns. For example, comparing 4KB vs. 512KB curves in Figure 9.6a, we see that only about 90% of the messages for the 4KB buffer complete in under 10 ns whereas 99.8% of the messages complete in under 10 ns for 512KB buffers. It also appears that this effect is most dramatic for small buffers, and the benefits of having a larger buffer drops off after about 512KB (i.e. the performance characteristics of 512KB, 1MB, and 4MB are similar)[3].

After 10 ns, there are two sharp "knees" at around 50µs and 100+ milliseconds. The first knee is due to the background thread sleeping betweens rounds of polling the staging buffer. It disproportionately affects the smaller buffers, as the smaller buffers can completely fill up between sleeps. This causes more log messages to be delayed as the logging thread has to wait for the background

[3] I am unsure why this performance drop-off occurs. I observe the same drop-off even in a microbenchmark that only records Intel Timestamp Counter [29] readings to a contiguous byte array in memory (the *rdtsc* line in Figure 9.6a). This suggests the drop-off is caused by something outside the application's control.

thread to wake up, process the log messages, and free more staging buffer space. The second knee at 100+ milliseconds is due to I/O. In scenario, the background thread runs out of space in the I/O buffer used to store compressed log data (64 MB for this test) and has to wait until the I/O completes. This results in a significantly longer delay, but affects fewer log messages.

One interesting observation is that the more frequent the stalls, the better the worst-case performance. This can be seen in Figure 9.6a. Although the 4KB buffer experiences a 99.5% tail latency within 10^5 nanoseconds vs. the 99.9999% for the 4MB buffer, the slowest operation for the 4KB buffer only takes tens of microseconds vs. the hundreds of milliseconds for the 4MB buffer. The reason for this is because the smaller buffers cause the logging thread to stall more often due to running out of staging buffer space. These more frequent, but short stalls give the I/O more time to complete. This results in better worst-case performance for the 4KB line. In contrast, the 4MB buffer stalls less frequently. This means a few messages have to wait a comparatively longer time for I/O to complete. Said another way, slowing down the logging also lowers the worst case I/O latency, because the I/O has more time to complete before the system runs out of buffer space.

Message size also plays an important part in determining how often delays will occur; the larger the message size, the more often delays will occur. We can see this by comparing the curves between Figure 9.6a and Figure 9.6b. In all cases, the curves in (b) appear ”higher” in the graph than (a). This indicates that a larger fraction of the messages are taking longer to process. For example, for the 4KB line, we see that 90% of the log messages complete within 10 ns for the *staticString* case (Figure 9.6a), but only about 60% do the same for the *complexFormat* case (Figuree 9.6b). This difference is due to message sizes. The *staticString* requires only about 3 bytes to represent vs. 19 bytes for *complexFormat*. The larger message sizes of (b) fill up the buffer faster, which result in fewer messages being buffered before the logging thread has to stall and wait for the background thread. This causes the entire graph to shift up.

Overall, one should set the size of the staging buffers based on the desired tail performance characteristics. Smaller buffers tend to cause more stalls, but can result in better worst-case latency in the steady-state if log messages are small. Conversely, larger buffers can allow the system to buffer larger bursts of log messages before experiencing I/O delay. NanoLog sets its default buffer size to 1MB, where the system appears to see diminishing returns; i.e. increasing the buffer size further does not appear to improve the fraction of log messages completing in under 10 ns.

One **very important caveat/note** is that this benchmark is a bit unrealistic/contrived. It assumes that the application is logging as fast as possible without performing any work between each log statement. This results in the buffers filling up and the logging thread blocking on I/O, which results

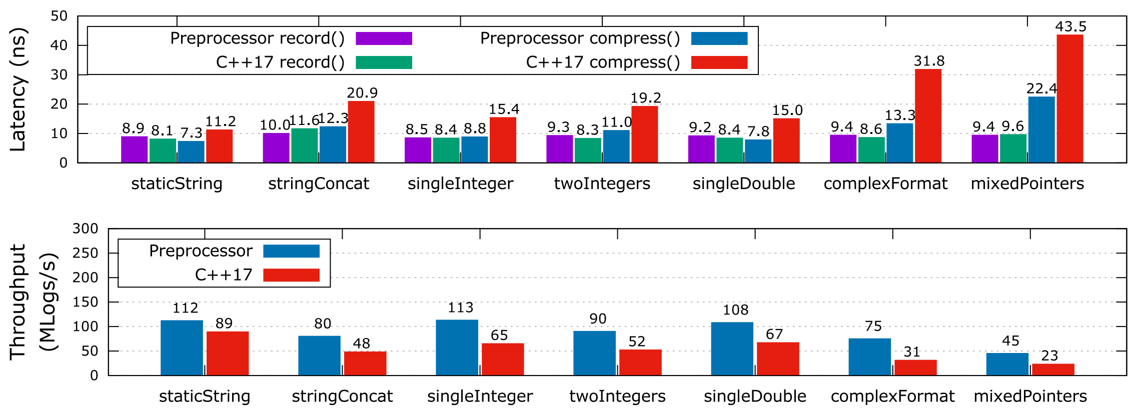


Figure 9.7: The performance of C++17 vs. Preprocessor NanoLog on average `record()` and `compress()` function execution times (top) and overall throughput (bottom). In each experiment, a single message was logged 100M times back to back with no delay. The x-axis shows the log messages used and the first six labels correspond to the messages in Table 9.3. The rightmost message, *mixedPointers*, was specifically added to this benchmark to stress the C++17 templated `compress()` function and alternates pointer and string format specifiers i.e. `NANO_LOG(``%p %s %d %s %p', 'abc', 'abc', 50, 'abc', 'abc')`. Each system was measured with the log file set to `/dev/null` to demonstrate the performance without disk bottlenecks. Furthermore, the average latency measurements (top) were performed with a single logging thread, while the throughput varied the number of logging threads between 1-8 and recorded the maximum throughput achieved. The results from this figure differ from Figure 9.2 as disk I/O was removed, and they differ from Table 9.4 as this figure reports average invocation latency vs. median/tail percentiles.

in the millisecond delays. For applications with a more sane logging rate (say 1 message per 1000 ns or 1M log messages/second), I expect the extreme tail to be significantly better and closer to the *rdtsc* line in Figure 9.6a[4].

9.7 C++17 vs. Preprocessor NanoLog Performance

To compare the performance of C++17 NanoLog with Preprocessor NanoLog, I ran a microbenchmark with a single thread repeatedly logging one of seven log messages back to back with no delay. I then measured the maximum throughput and average invocation latency for the generated `record()` and `compress()` functions with the log file piped to `/dev/null` to eliminate disk

[4]There can still be delays upwards of 10^5 nanoseconds for reasons outside the application's control.

interference. Figure 9.7 shows the results.

The first thing to notice is that C++17 NanoLog is not as performant as Preprocessor NanoLog. The `compress()` functions it generates are consistently 1.5-2.4x slower than the preprocessor counterparts, with the worst case being when the log message alternates pointer and string types (*mixedPointers*). There are two reasons for this slowdown. First as mentioned in Section 4.4.2 of Chapter 4, the C++17 `compress()` function must perform a small amount of computation at runtime to differentiate between pointer and string types. In contrast, preprocessor NanoLog requires no such logic. This is why the largest performance gap occurs with *mixedPointers*; it alternates logging pointer and string types. Second, the C++17 front-end generates `compress()` functions that are not easily in-lined or optimized by the compiler. The C++17 front-end specializes the `compress()` function within the `record()` function and stores a pointer for later invocation. Since the compiler can neither "see" the arguments nor in-line the functionality at specialization time, it cannot make any optimizing assumptions and generates more conservative, slower code.

Conversely, the `record()` functions generated by both systems are within a 10% or 1 ns margin of each other. The performance is similar here because the C++17 `record()` function is specialized and used in the same code location. As a result, the compiler can make optimizing assumptions on the arguments and in-line the logic, resulting in similar performance as preprocessor NanoLog.

Lastly, the throughput of the system is ultimately determined by disk bandwidth or the slower of the two generated functions. In Section 9.1.3, the two systems generated similar throughput performance since they were both bottlenecked by disk performance (250MB/s). However, with the disk bottleneck removed for Figure 9.7, performance is dictated by the slower of the `record()` and `compress()` functions. This results in a throughput that is roughly the inverse of the slower operation. Preprocessor NanoLog dominates throughput in this scenario since the `compress()` function is typically the slower operation, and Preprocessor NanoLog generates faster `compress()` functions.

As a side note, one may also notice that there is a base cost for `compress()` even when the log statement has no arguments (staticString). This is an artifact of NanoLog's architecture. Both versions of NanoLog invoke the `compress()` function by dereferencing a function pointer. This operation tends to cost 7-8 ns. Thus, even though the end function is effectively a no-op, it still requires 7-8 ns to dereference the no-op and execute it.

Overall, Preprocessor NanoLog is more performant than the C++17 version, as it generates more optimized `compress()` functions.

9.8 Concluding Remarks

Overall, NanoLog is a highly performant system. It offers a logging throughput higher than other state-of-the art systems at dramatically lower latencies. Even when integrated in a logging project with a realistic workload, the system still outperforms its competitors. Additionally, we've seen that the NanoLog binary log offers performance benefits in log analytics applications, providing up to a 10x improvement on processing throughput. We've also measured one level deeper to break down the performance of individual components and identified disk bandwidth as one of the major culprits hindering performance. The only downsides to NanoLog are that it emits a binary log that needs to be post-processed and slightly increases the application's binary size and compilation time.

Chapter 10

Extensions and Limitations

NanoLog is not a perfect system and has limitations. In this chapter, I will describe some ways in which one could improve NanoLog to address some of its limitations as well as list other limitations that I believe are fundamental and cannot be fixed.

10.1 Extensions

The NanoLog system in its current form on GitHub [70] is a proof-of-concept implementation for C++, and I've listed a few ways in which the system can be improved or extended below.

10.1.1 Support for Additional Programming Languages

As of the writing of this dissertation, the NanoLog system only supports C++ applications. However, I believe the ideas and portions of the NanoLog system can be generalized to support any language that exposes its source code. For example, the preprocessor component can be modified to recognize the syntax of different languages and perform code injections. The runtime could be either linked in via bindings (such as through JNI for Java) or a new runtime library can be developed for each additional language[1]. The post-processor component can be reused without modification, so long as the language-specific runtimes output a compatible format.

Furthermore, I suspect the ideas of NanoLog can be directly implemented in popular compiler backends such as LLVM [32]. The background thread and NanoLog runtime would be absorbed

[1]The runtime can be fairly rudimentary as it only needs to provide storage for log statements and a mechanism to persist them to disk.

into the system library, and the NANO_LOG () transformations into more optimized code can be done by the compiler. This move would also make it fairly trivial to port the NanoLog system to even more languages, as one would only need slight modifications to the compiler front-ends to recognize NANO_LOG () invocations.

The only limitation to NanoLog's ideas and generalizability is that it would only be capable of optimizing NANO_LOG statements that follow a strict printf-like specification with a string-literal format string. In other words, it would not be able to optimize any log statements that are dynamically generated and evaluated (such as with JavaScript's `eval()` [11]). It would also not be able to optimize any log statements that log more complex data types than the ones supported by the printf specification [6]. Overall, this limits the NanoLog system to one very specific style of logging.

However, I believe that NanoLog's ideas can be ported to most programming languages.

10.1.2 Remote Logging and Analysis with Other Frameworks

The NanoLog system currently only supports logging to the local disk, and the NanoLog aggregation API it offers is quite limited compared to third party log analysis frameworks like Splunk [57]. Fortunately, the design of the NanoLog system is amenable to streaming the log data in a compressed format over the network to third party log storage/analysis systems. These systems can then perform the analysis/inflation of the messages remotely. This method saves on network I/O as the log messages remain in their compressed format, and in certain cases, reduces the cost of parsing and indexing the log data. In this section, I will describe at a high level (a) the modifications that need to be made to the NanoLog system to support streaming the log data over the network, and (b) a hypothetical implementation of the receiving analytics engine on top of Splunk.

To stream NanoLog log data directly over the network, the NanoLog background thread requires two modifications. First, after the plumbing is set up to connect to an external host, the background thread should be modified to stream compressed *buffer extents* to the remote log processing framework. The buffer extent structure is chosen as the basic building block, because it is already a well encapsulated chunk of dynamic log data with a size field. Second, the background thread should also be modified to accept incoming requests for the dictionary. This latter call is needed because the dynamic log data encoded in the buffer extents cannot be interpreted without a dictionary. Thus, if the remote processing framework ever loses the dictionary via a crash, host migration, or cold start, it must be able to recover the dictionary from the NanoLog application.

On the remote analytics framework end, the framework needs to be able to maintain/query the NanoLog dictionary in a fashion similar to the post-processor (for example with the dictionary entries in an indexable array) and be able to accept log data. Once the log data arrives, the framework can use the same techniques as the post-processor to lookup the log message's unique log identifier in the dictionary, find the associated format string, and parse/interpret/index the arguments. The log data can then be placed into the framework's internal indexes without formatting (if the system supports it) and only format the messages when the user views them. If the analytics framework ever encounters a log message that it does not have the associated dictionary entry for, then it should query it from the NanoLog application.

One additional feature the remote system needs is a way to manage sessions. More than likely, there will be multiple applications using the NanoLog system. Each instance of the application will have its own specific dictionary to interpret its log statements, and the dictionaries won't be interchangeable. Thus, the system needs to handle mapping a specific dictionary with every connection to a NanoLog application. This can be trivially done by maintaining sessions to the applications (via persistent TCP sockets or cookies) and querying for the dictionary whenever a new session is created.

To demonstrate the feasibility of this setup, I will describe hypothetical implementation involving Splunk [57] as the log processing framework. Splunk is a popular, log processing framework that allows for searching, indexing, and analytics to be performed on log traces of any kind. However, to get the most of the system, it is often recommended that the log sources match one of the default data sources it can natively parse (such as syslog or Windows Events), or create an "Add-On" to perform custom processing on the incoming data.

To process the log data in the Splunk platform, I would build/utilize a custom "Add-On". For the purposes of demonstrating feasibility, I would start with the Protocol Data Inputs Add-On [7]. This Add-On already implements a framework that allows for custom processing of incoming binary data via Python/Javascript scripts before passing it to the Splunk indexer. To to get a proof-of-concept going, I would write a custom NanoLog log file parser in Python/Javascript and perform the decompression of log statements within the Splunk framework. This configuration would allow the NanoLog system to save network bandwidth and perform inflation on a remote system. To obtain even higher performance gains, one could write a custom Add-On that would not only allow for decoding of the binary log arguments on ingest, but also defer formatting until the user views it in the Splunk application. To be performant, this custom Add-On would only partially inflate the log messages' arguments such that they're indexable by the Splunk engine and then implement a custom "Dashboard" that would only generate the full, human-readable log statements when viewed

```
enum State
{
        State1, // State 1
        State2, // State 2
        State3, // State 3
        ....
}

const char* enumToString(enum State s)
{
        switch(s) {
        case State1: return "State 1";
        case State2: return "State 2";
        case State3: return "State 3";
        ....
        }
}

void logStatus(enum State s) {
        // Normal Print Statement
        NANO_LOG("The current state is %s", enumToString(s));

        // Stream Print Statement
        cout << "The current state is" << enumToString(s);
}
```

Figure 10.1: Pseudo C++ code that demonstrates a common method in which enums are logged as static strings and how stream operators can also contain static string data.

on-demand.

Overall, NanoLog can be modified to stream its log data. It only needs the additional ability to connect to a remote host, send over buffer extents, and accept requests for dictionary queries. Furthermore, external log processing platforms like Splunk can be modified to accept NanoLog log data.

10.1.3 Extracting Enums

Another avenue for improving NanoLog's performance is to add the ability to extract pretty-print strings for C++ enumerations. In C++, enumerations are a method of allowing developers to map symbols (i.e. names) to integer values. These symbols typically only have meaning within the application source itself, and the compiler will transform all references to integer values. Thus, to make sense of the values when logging, developers often write pretty-print functions that take in the value of an enumeration and output a human-readable string which they can then log with the "%s" specifier. Figure 10.1 shows an example. This method makes it easier for developers to read enumeration values in the log, but it is suboptimal for NanoLog, as NanoLog does not apply any compression to strings and will waste I/O outputting the full string every time.

NanoLog can improve upon this short-coming by applying a second level of extraction for enumerations. More specifically, the preprocessor can be modified to detect where enumerations are logged, extract the static pretty-print string used by the enumeration, store the mapping of enumeration values to static strings, and replace the code to log the raw value of the enumeration instead. The post-processor can then be modified to read in the mapping of enumeration values to static strings and perform the replacement before formatting the log message. This method would make the runtime more efficient as the application would (a) log a smaller integer type rather than a string, and (b) use its variable length encoding compression scheme to further reduce the I/O size.

The hardest part of implementing enumeration support would be finding the mapping of enumeration values to pretty-print strings. While I don't have a concrete solution myself, I do have some partial ideas of how one may be able to do this. One solution is to detect enumeration definitions, use the enumerator names/values (i.e. `State1, State2, State3` in Figure 10.1) as the static string descriptor, communicate this information to the post-processor, and have the post-processor use these values as the human-readable strings. Another method may be to detect pretty print functions with some heuristic (i.e. identify functions that take an enumeration parameter and output a static string), compile them into the post-processor instead, log only the values at runtime, and then have the post-processor invoke the pretty-print function instead. A final method may be to require users to define enumerations in a well documented way (such as having the pretty print name in comments after the enumerator name/value as in Figure 10.1) and having the NanoLog system extract these values for the post-processor. All these suggestions are partial solutions, and they all have different flaws. Nonetheless, I included these here in the hopes that someone will come up with a better solution and contribute to the NanoLog GitHub repository [70].

10.1.4 Stream Logging

In my exploration of the logging space, I also found that developers often like to use stream operators in-lieu of printf-style logging. In fact, some of the more popular logging libraries such as glog by Google [21] introduce stream logging as the default method of logging. Another example of streams' prevalence may be how many C++ "Hello World" tutorials start by teaching developers to log to use the "cout" stream to write to standard out. Clearly, streams are a common way developers log, and it would be beneficial if NanoLog supported streams.

NanoLog currently does not support stream operators, however I believe that it is possible. For example, in the last line of Figure 10.1, we can clearly see that the user outputs a well defined, human-readable static string followed by an invocation to the pretty print function. One can build a NanoLog-like system that would extract the static text, replace it with an integer or enumeration value, output the integer value to the stream instead of the static text, and use a post-processor to reinterpret the stream of numbers. This setup would not be unlike how NanoLog currently operates, except that the new system would search for NanoLog-defined stream classes and replace the static text in between the `<<` operators with integer mappings instead. One limitation of this setup when using the current NanoLog techniques is that one may not be able to statically detect the type of the arguments passed to the `operator<<` without compiler help. Thus, NanoLog would either have to identify the arguments dynamically and note them in the log file, or it would need to integrate with a compiler and ask the compiler for hints.

Nonetheless, I believe that with some effort, one can modify the NanoLog system to also support streams.

10.2 Fundamental Limitations

In addition to the fixable limitations, NanoLog also has four fundamental limitations that are likely not fixable by further extending the system: NanoLog introduces extra complexity into an application, can use more resources than a traditional logging system, is restricted to a specific style of logging, and does not fully solve the problem of visibility in applications.

One of the biggest limitations of NanoLog is that it introduces extra complexity into an application's development cycle. In particular, NanoLog's non-conventional design requires users to integrate a preprocessor in their compilation chain[2], to store intermediate, binary representations of log messages, and utilize a post-processor to re-inflate the logs into something consumable by humans or other machines. Compared to traditional logging systems where users only need to link against the library and the runtime will automatically generate human-readable log messages, NanoLog adds extra steps and extra levels of complexity. This makes NanoLog potentially more difficult to integrate into existing environments and adds additional points of failure.

Furthermore, if the goal of the user is to obtain human-readable log messages, NanoLog actually increases the end-to-end consumption of resources. In particular, the NanoLog front-end performs additional analysis and inflates the binary size and compilation time by about 2-19%, the runtime utilizes extra CPU and I/O to emit an intermediate binary log file, and the post-processor must

[2]...or use a more recent compiler for C++17 NanoLog.

expend additional I/O to read back the binary log file and output it in a human-readable format. In other words, NanoLog's deferral of formatting log messages creates intermediate representations that cost extra resources to produce and maintain. Thus, NanoLog only makes sense in situations where runtime performance is paramount and either extra resources can be dedicated to the post-processor or it is uncommon for humans to consume the log.

Third, NanoLog relies on a strict adherence to a single type of logging style; it requires the user to have printf-style log statements with string-literal format strings. The NanoLog system exploits the printf format string as a specification for the order and type of dynamic arguments that will be passed into the system. In situations where this data is not readily available at compile-time (such as with dynamic format strings), then NanoLog cannot extract the static information, and it cannot infer the type/order of the log arguments to generate specialized functions. NanoLog would instead have to fall back to outputting the dynamic format string and parsing it at runtime to infer the argument data types[3], negating most of the bandwidth and compute gains. Furthermore, NanoLog's strict adherence to printf-style logging means that it cannot support other forms of logging such as outputting data structures (lists, maps, arrays, etc). In other words, NanoLog's techniques are only applicable to a very specific definition of logging.

Lastly, NanoLog is not a silver bullet to application visibility; it does not solve the problem of what/when/where developers should log. NanoLog only focuses on making the logging operation cheaper and does not attempt to make any suggestions on how to use this operation. It is entirely possible for developers to saturate NanoLog's throughput without gaining any additional visibility into the application through suboptimal placement of log statements. Furthermore, with an increased rate in logging, users will have to sift through more raw data in order to find interesting events. This may have the adverse effect of making an application more difficult to understand as the user is inundated with superfluous information. Thus, while NanoLog attempts to address application visibility by allowing developers to log more often, it still relies on the developers to log the right things at the right moments.

[3]C++17 NanoLog can infer the order/type of the arguments using variadic templates, and it can build the dictionary of static information at runtime. However, it would still need to parse the format string at runtime and expend I/O to output the dynamic format string (at least once per new string).

10.3 Summary

In short, the NanoLog system can be extended to support other programming languages, log remotely, extract additional static information from enumerations, and support stream logging. However, regardless of modifications, NanoLog will always increase the complexity of development environments, use more resources to produce human-readable log messages, and rely on extracting static information at compile-time. It also does not directly solve the problem of gaining visibility into low-latency applications. However, even with this set of limitations, I still do believe that NanoLog contributes to the state-of-the-art for logging and will speedup logging for most developers in most environments.

Chapter 11

Related Work

The NanoLog system shares many ideas and techniques found in other software systems. In this chapter, I will discuss how NanoLog is similar to three broad categories of software: traditional logging systems, software tracing packages, and remote procedure call (RPC) or serialization libraries. Then, I will take a deeper dive into two specific software systems most similar to NanoLog: Event Tracing for Windows [47] (ETW) and KUTrace [54]. Finally, I will conclude on work that is more tangential to NanoLog itself, but related to the broader theme of increasing visibility in software systems.

11.1 Traditional Logging Systems

At first glance, the NanoLog system is perhaps most similar to traditional logging systems. Like these systems, NanoLog exposes a familiar printf-like API to allow developers to specify arbitrarily complex log messages, and the system can emit a full human-readable log file. Other logging systems also include performance-enhancing designs such as buffering intermediate log messages at runtime and utilizing a background thread to perform expensive I/O. However, these are only surface level similarities; NanoLog employs techniques that changes how one interacts with the system and greatly enhances its performance relative to traditional logging systems.

The greatest difference between NanoLog and traditional logging systems is that it introduces extra steps at compile-time and post-execution. NanoLog adds a preprocessor to analyze log statements and generate highly optimized, log-specific code to be injected into the application at compile-time, and NanoLog utilizes a post-processor to defer formatting of log messages and emits a binary log file at runtime. This contrasts with traditional logging systems that are only

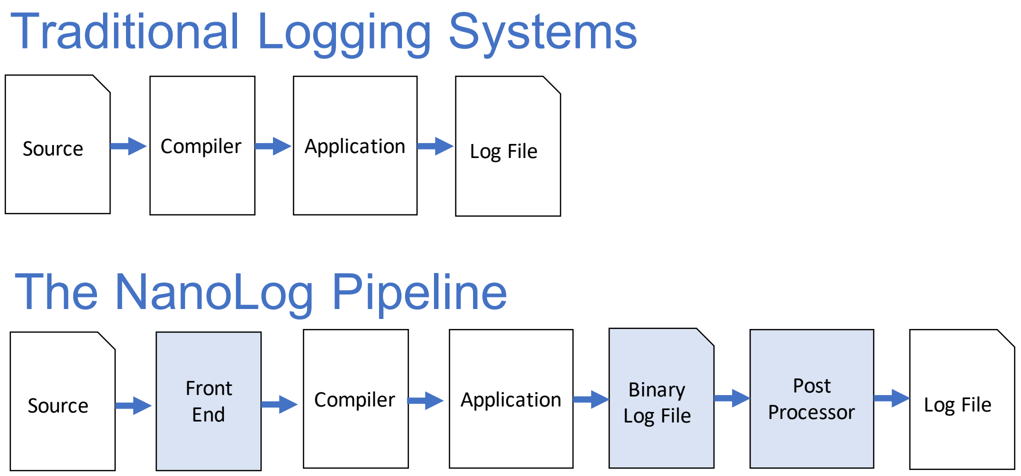


Figure 11.1: The pipeline from source file to human-readable log for traditional logging systems (top) vs. NanoLog (bottom). For traditional logging systems, users can expect their sources to be directly compilable, and the resulting application will output the log file in a human-readable format. In contrast, the NanoLog pipeline adds three new components: sources must first pass through the *front-end* (i.e. either the preprocessor or the C++17 templating engine) before being compiled, the application outputs a *binary log file*, and the *binary log file* must be passed through the *post-processor* to obtain the full human-readable log file.

linked as libraries at compile-time and emit the full human-readable log file as soon as the application executes. NanoLog's optimizations complicate the deployment strategy relative to traditional logging systems, as the user must integrate two additional components into their software stacks (Figure 11.1). However these optimizations propel the performance of NanoLog far beyond that of any traditional logging system.

Overall, while NanoLog exposes a similar user interface as traditional logging systems, its internal architecture, usage, and performance characteristics are very different.

11.2 Tracing Software

Given NanoLog's ultra-low, nanosecond scale logging mechanisms, comparisons have been drawn between it and highly specialized tracing software such as PerfUtils [49], kutrace [54], and wait-free queuing [40]. These systems allow the user to record when certain code segments (events) execute in the application and are primary used for performance debugging. They include a host of optimizations similar to NanoLog to achieve nanosecond scale performance, such as using high performance CPU counters [29] to gather timing information, using lockless, in-memory ring buffers [40] to store events, and minimizing the number of bits required to represent each event. How they differ from NanoLog is that they can typically only sample events and they limit the amount of information recorded.

Tracing systems achieve their high performance by limiting the number/types of arguments per event and sampling events. Tracing systems are often so high performance that most of their events cannot be persisted to disk in a timely matter. Thus, they only record a burst of information into an in-memory ring buffer and then pause to output the information. For example, PerfUtils::TimeTrace will only output the last 8192 events whenever `print()` is invoked [49], and KUTrace will only record events for 30-120 second intervals at 200k events per second [54]. Furthermore, the amount of information recorded by each event is severely limited; this is done to minimize the byte size of each event and maximize the limited space space in the ring buffer. For example, PerfUtils::TimeTrace will only record up to four 4-byte integers per event and even more extreme, KUTrace will only record up to 3-bytes of arguments per event. In contrast, NanoLog is an always-on, guaranteed logger, will never drop a log message, and can include a virtually unlimited number of arguments.

The flip side of NanoLog's design is that it can induce higher performance variation in the application when compared to traditional tracing systems. This variation stems from three design differences. First, NanoLog utilizes a separate thread to compress the log statements and output them to disk. This extra thread utilizes extra resources that may cause cache or thread migration interference with the application. Second, NanoLog's guaranteed logging means that if log messages are generated faster than they can be output, the system will block, resulting in performance variation. Tracing systems do not suffer from this problem as they typically utilize a ring buffer and will simply overwrite older events when the application records faster than can be output. Lastly, NanoLog's acceptance of more varied or general log arguments means that each statement can take a variable amount of time (for example, a 100-byte string would take longer to output than a 4-byte integer). Tracing systems do not suffer from this problem as their arguments are more restricted and fixed-size. Overall, NanoLog has a potential for higher performance variation than traditional tracing systems.

Fortunately, NanoLog can emulate much of the consistent and high performance of tracing systems with minor modifications. More specifically, NanoLog can disable its background thread to reduce thread interference, only persist events to disk once a sample period is over, increase the staging buffer sizes to match the ring buffers of other tracing systems, and limit the number of allowable arguments in a single log statement[1]. With these modifications, the NanoLog system operates nearly identically to the tracing systems. The only difference after these modifications is that NanoLog will still require more memory for the same number of events, as its in-memory representation is not as compact as some other systems.

Overall, tracing systems offer more consistent performance than NanoLog, but they come at the cost of requiring sampling and limited arguments.

11.3 RPC Systems

Lastly, the code generation techniques used by NanoLog are similar to low-latency RPC/serialization libraries such as Thrift [55], gRPC [22], Google Protocol Buffers [65], and Cap'n Proto [67]. The goal of these systems is to transmit messages between processes with the fewest number of bytes possible. To do this, they use static message specifications (*schemas*) to name symbolic variables and their types (Figure 11.2) and use a special compiler to generate logic to encode/decode the data in a succinct format for the application to use. This methodology is not unlike how NanoLog utilizes a log message's format string and a compile-time front-end. In particular, the NanoLog system uses the formal specifications for a printf format string [6] to encode the order and allowable types for a log statement's arguments and uses a special preprocessor to generate logic to encode/decode that data.

Another similarity between the systems is the use of variable length encoding for integers as a form of compression. Cap'n Proto [67] in particular came to the same conclusion as NanoLog that most integer values transmitted/logged are small. As a result, both systems independently decided that variable length integer representations are the best form of compression for their data types. The exact encoding differs between systems [66], but the core principle is the same.

RPC systems differ from NanoLog in that they operate in a different domain, are typically cross-language, and can communicate more data types than NanoLog. In particular, RPC systems operate in the domain of networked communication between processes, whereas NanoLog aims to communicate log messages to human users. Since the endpoint is other computers, RPC systems can represent all their data in a binary-only format and support more complex data structures like queryable lists, dictionaries, and C++ structs; NanoLog is limited to logging basic data types. RPC

[1]Experiments have shown that NanoLog can maintain its <10 ns response latency even when 5 or more integer arguments are logged in a single message.

```
@0xdbb9ad1f14bf0b36;  # unique file ID, generated by `capnp id`

struct Person {
  name @0 :Text;
  birthdate @3 :Date;

  email @1 :Text;
  phones @2 :List(PhoneNumber);

  struct PhoneNumber {
    number @0 :Text;
    type @1 :Type;

    enum Type {
      mobile @0;
      home @1;
      work @2;
    }
  }
}

struct Date {
  year @0 :Int16;
  month @1 :UInt8;
  day @2 :UInt8;
}
```

Figure 11.2: An example schema file for the Cap'n Proto RPC system copied directly from the online documentation [68]. This schema shows how users of the RPC system can define C++-like structures that indicate a symbolic name for a variable, the order in which it appears within the structure (as indicated by the numbers after @ symbols), and the type of the variable (i.e. Text, Date, List(...)). This definition is then compiled by the Cap'n Proto compiler to generate source code that serializes and deserializes the information, much like how NanoLog uses the printf specifications [6] to generate specialized functions to serialize/deserialize log information.

systems are also more general and are built to operate across many different languages whereas NanoLog is currently only implemented for C++ applications.

NanoLog's limited scope does present usability and performance advantages. First, NanoLog's use of the existing *printf* specification [6] within the C++ language makes it easier to use. Unlike the other systems, NanoLog does not require one to learn a new domain-specific language for the schema, and it does not require the user to explicitly maintain a separate schema file to be used with the sources. Instead, users simply write code as they would for a normal C++ printf statement and compile. Second, NanoLog's restrictions to a single language and more limited scope for argument types allow for higher performance. Since NanoLog is limited to C++ (or more generally a single language at a time), it can output data in a format most convenient for that language. In contrast, RPC systems may need to perform conversions between languages or to a more general representation for wire transport. As a result, the serialization/deserialization costs in most RPC libraries are significantly higher than NanoLog's more tightly scoped argument types.

Overall, the goals and techniques used by NanoLog and RPC systems are similar in flavor, but are applied to different domains with different performance envelopes.

11.4 Event Tracing For Windows

Event Tracing for Windows (ETW) [47] with the Windows Software Trace PreProcessor (WPP) [28] deserves special attention as it is most similar to NanoLog. This system was unbeknownst to me when I designed NanoLog, but WPP appears to use compilation techniques similar to NanoLog. Both use a preprocessor to rewrite log statements to record only binary data at runtime and both utilize a post-processor to interpret logs. However, ETW with WPP does not appear to be as performant as NanoLog; in fact, it's on par with traditional logging systems with median latencies at 180ns and a throughput of 5.3Mop/s for static strings. Additionally, its post-processor can only process messages at a rate of 10k/second while NanoLog performs at a rate of nearly 500k/second (I was not able to ascertain why ETW is slower than NanoLog given its closed-source nature).

As far as I can discern, there are five main differences between ETW with WPP and NanoLog: (1) ETW is a non-guaranteed logger (meaning it can drop log messages) whereas NanoLog is guaranteed. (2) ETW logs to kernel buffers and uses a separate kernel process to persist them vs. NanoLog's in-application solution. (3) The ETW post-processor uses a separate trace message format file to interpret encoded log files, whereas NanoLog uses dictionary information embedded directly in the log file. (4) WPP appears to be targeted at Windows Driver Development (only available in WDK), whereas NanoLog is targeted at applications. Finally, (5) NanoLog is an open-source library [70] with public techniques that can ported to other platforms and languages while ETW is proprietary and locked to Windows only. There may be more differences than what I've listed (such as the use of compression). However since ETW is closed source, I can only list what is documented.

Overall, ETW uses similar techniques to NanoLog, but it is closed-source and not as performant.

11.5 KUTrace

KUTrace [54] is a Linux kernel tracing utility that uses some of the same techniques as NanoLog and exhibits similar performance. KUTrace aims to instrument every kernel/userspace transition for about a minute or two at a time and patches the kernel at the source level to do so. Its goal is to provide a level of instrumentation that could help demystify 99th percentile latencies in unmodified applications. It is similar to NanoLog in that it achieves a similar average performance as NanoLog (about 12.5 ns per event), emits a binary log file, and utilizes a post-processor to make sense of the binary log file. They both also share many of the same runtime optimizations such as using a low-level instruction, *RDTSC* [29], to gather timing information and recording events to a lock-less circular buffer [40]. However, there are three primary differences between NanoLog and KUTrace.

First, KUTrace was designed to sample trace data only. In particular, it was designed to burst record all incoming events for 30-120 seconds into an in-memory buffer and then stop all recording to process the event data and persist it to disk. NanoLog, on the other hand, was designed as a persistent, always-on logging system. The trade-offs between the two systems is that KUTrace will never block an application from executing (as it will simply drop new events once the buffers are full), whereas NanoLog can block, but will never miss any critical events. This blocking can result in more variable performance when compared with KUTrace.

Second, KUTrace is extremely limited in the information it collects. KUTrace events are highly specialized to trace function call events and are thus optimized to fit within a single, 8-byte word. This means that after accounting for timestamps and an event identifier, each event only has 3 bytes remaining to store arguments. These three bytes are used to save the first two bytes of the first function argument and the first byte of the return value. NanoLog, on the other hand, allows for as many arguments as the system's built-in *printf* function would allow for, and this means NanoLog's log messages can contain more semantic information than KUTrace. The trade-off here is that KUTrace has consistent message sizes and thus consistent performance, whereas NanoLog's performance can vary with the number of arguments.

Third, KUTrace does not use an automated process to generate event identifiers. For every event the user wishes to record in KUTrace, the user has to first define a unique C++ *enumeration* name and value, pass the value to the `record()` function equivalent, and manually add the mapping into the post-processor. In my opinion, this process is extremely error-prone (the user has to consistently edit 3 lines of code in three different files for every line of instrumentation), and it reduces the usability of the system. In contrast, the NanoLog system uses a preprocessor or front-end to inject a unique log identifier for every log statement and communicates this information to the post processor automatically; the user only has to write one line of code to log a message. In my opinion, this makes NanoLog much easier to use than KUTrace.

Lastly, here are a collection of minor, miscellaneous differences between NanoLog and KUTrace. KUTrace runs in the kernel whereas NanoLog runs purely in userspace. The benefit of the former is that KUTrace can turn off preemption during the recording operations whereas NanoLog cannot. KUTrace also has a graphical viewer for events, allowing users to view spans or how function calls nest. NanoLog currently does not have such a viewer.

Overall, the biggest similarity between KUTrace and NanoLog is their performance envelopes, but the two systems are designed to serve different purposes. KUTrace is optimized for short(ish) bursts of event tracing whereas NanoLog is always-on logging. KUTrace is optimized primarily to trace function calls, whereas NanoLog can be used to instrument a larger variety of software constructs. And lastly, NanoLog is easier to use than KUTrace as NanoLog uses a preprocessor or front-end to automatically create and map event identifiers in the code. The consequence of NanoLog's features, however, is more varied performance, as it may block for free space or consume more resources to represent more arguments.

11.6 Related Areas of Research

Logging plays only a small part in the grand scheme of increasing visibility into low latency systems.

Moving beyond a single machine, there are also distributed tracing tools such as Google Dapper [51], Twitter Zipkin [74], X-Trace [16], and Apache's HTrace [60]. These systems operate in distributed settings where a software event (such as a user request) can propagate over thread, process, machine, and even data center boundaries. In these settings, simply logging on a single machine is not sufficient to fully understand the end-to-end performance of a system. Thus, distributed tracing tools are utilized to track the causality of log messages related to a single request[2] and link them together for later analysis. This provides the user with a full end-to-end picture of how a request propagated through various software components. While these systems are great for understanding distributed behavior, they do not accelerate core logging performance like NanoLog.

After producing log files, a developer often has to sift through a mountain of log data to find anomalous behavior or interesting events. To aid in this effort, multiple online machine learning and database services have emerged to collect log messages and provide insights such as Splunk [57], Datadog [8], DISTALYZER [41], and others [43]. These systems typically work by ingesting raw human-readable log messages, performing various transformations or statistical analysis on the data, and then presenting the user with an interface that either highlights anomalous events (such as longer-than-expected requests) or allows the user to perform SQL-like queries to narrow the list of interesting events. These systems enhance a user's ability to "see" the behavior of a system and quickly identify problem spots. NanoLog hopes to increase the efficiency of these systems by presenting the log files in a compressed, binary format rather than the full human-readable log messages to save I/O and processing time.

In addition to the static log messages I've discussed so far, there are also systems that employ dynamic instrumentation [26] to gain visibility into applications at runtime such as Dtrace [25], Pivot Tracing [36], Fay [12], and Enhanced Berkley Packet Filters [24]. These systems allow a developer to insert log new log messages into a system *while the system is executing*. This contrasts with conventional logging where log messages are determined a priori at compile-time and remain unchanged until the application is modified and recompiled. This alternate method of logging enables faster debugging iteration as a developer can inspect a system live, quickly add new log messages, and repeat until the bug is found without ever shutting down the system. The downside to this approach when compared to more traditional logging methods like NanoLog, is that instrumentation must already be in place in order to initially detect problems or enable post-mortem debugging. As a result, I imagine future developers will use a synergy of extremely fast logging to provide an operational baseline and only add new log statements with dynamic tracing when slowness or bugs are encountered.

[2]They do this by propagating and logging a unique identifier for each request in the system.

11.7 Remarks

The NanoLog system shares techniques with a variety of software systems, but it brings to the table a unique blend of high performance and flexibility. NanoLog inherits the API of traditional logging systems, the performance characteristics of tracing systems, and the code generation techniques of RPC/serialization systems. Moving forward, I imagine NanoLog's techniques may find their way into distributed tracing tools and log analysis engines and synergize with dynamic instrumentation tools. NanoLog alone does not solve the issue of limited application visibility, but it plays a part by providing low-latency logging.

Chapter 12

Conclusion

NanoLog is a highly performant, nanosecond scale logging system that shifts work out of the runtime hot-path and into the compilation and post-execution phases of the application. It aims to increase visibility into low-latency, high performance applications by allowing developers to log more information while utilizing the same amount of resources at runtime.

Several key techniques allow for NanoLog's high performance:

- **Separation of Static and Dynamic Information**: NanoLog separates the static and dynamic log information and puts them on separate data paths. Static information is persisted just once in the log file, while the dynamic information is saved on every log invocation. This allows the system to dramatically reduce the amount of data flowing through the system at runtime.

- **Precomputed Logic**: NanoLog predetermines the logic that needs to occur for each log statement at compile-time. It generates the logic to record and compress a log statement's arguments to disk and compiles it into the application. This allows the application to execute highly optimized, in-line code instead of parsing the log statements at runtime to determine the logic.

- **Deferred Formatting**: NanoLog assumes that most log messages are never consumed by humans in the normal case, so it doesn't waste resources to format them at runtime. Instead, it outputs a binary log file and provides a separate, post-processor application to perform formatting.

- **Extremely Light Weight Compression** NanoLog compresses its log arguments with variable length integer encoding. This style of compression can be performed much faster and with fewer resources than traditional dictionary based systems. This compression allows the runtime to emit a smaller log file with minimal compute and enables fast log processing.

Experiments show that with these techniques, NanoLog is able to achieve a logging throughput of 80 million log messages per second at a median latency of 7-19 nanoseconds. This is significantly more performant than current state of the art systems that can only achieve a few million messages per second with latencies in the hundreds or thousands of nanoseconds. Furthermore, the binary log files produced by the NanoLog system are easier to consume by machines, making log aggregation and analysis cheaper.

Overall, NanoLog is an extremely performant system. It allows for an order of magnitude more log messages with the same amount of resources, and its printf-like API means it can be readily used today. An implementation for C++ is available on GitHub [70].

12.1 The Future of Logging

In my time in the instrumentation field, I've noticed a trend. Software is becoming so complex and distributed that it's difficult to understand from the standpoint of a single machine. As a result, large technology companies are investing more and more in distributed tracing infrastructures beyond a single machine, and multiple startups are banking on providing distributed tracing infrastructures to smaller companies that cannot afford to build the infrastructure themselves. All the signs point towards distributed logging as the future.

This is understandable, with the rise of micro-services and instant access to distributed infrastructures via cloud services, the need to move logging and instrumentation beyond a single executable, beyond a single, and in some case beyond a single data center is important. Knowing exactly what's happening with an application is key to driving down bugs, improving the user experience, and reducing waste in data centers. There's a lot of money to be made there.

As a result, I don't envision that the NanoLog system will become a staple of logging anytime soon, especially not the prototype implementation on GitHub. Instead, I hope that the ideas and techniques that I've explored would be integrated into newer systems; systems that integrate well with distributed tracing and provide value beyond a single machine.

12.2 Final Comments

With traditional logging systems, developers often have to choose between application visibility or application performance. However using the techniques of NanoLog, I hope that developers will be able to log more often and log in more detail, making the next generation of applications more understandable.

Bibliography

[1] Boost C++ Libraries. http://www.boost.org. 3, 95

[2] C++ REFERENCE. Condition Variable. https://en.cppreference.com/w/cpp/thread/condition_variable. 80

[3] C++ REFERENCE. Parameter pack (since C++11). https://en.cppreference.com/w/cpp/language/parameter_pack. 36

[4] C++ REFERENCE. std::max. https://en.cppreference.com/w/cpp/algorithm/max. 36

[5] C++ REFERENCE. Template argument deduction. https://en.cppreference.com/w/cpp/language/template_argument_deduction. 36

[6] CPLUSPLUS.COM. printf specification. http://www.cplusplus.com/reference/cstdio/printf/. 1, 3, 4, 6, 8, 13, 27, 32, 37, 63, 64, 65, 117, 127, 128, 129

[7] DALLIMORE, D. Sending binary data to Splunk and preprocessing it. https://www.splunk.com/blog/2016/07/28/sending-compressed-payloads-to-splunk.html. 118

[8] Datadog. https://www.datadoghq.com. 132

[9] DRAGOJEVIĆ, A., NARAYANAN, D., CASTRO, M., AND HODSON, O. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, Apr. 2014), USENIX Association, pp. 401–414. 2

[10] DWARF DEBUGGING INFORMATION FORMAT COMMITTEE AND OTHERS. DWARF debugging information format, version 4, 2010. 85

[11] ECMASCRIPT, ECMA AND EUROPEAN COMPUTER MANUFACTURERS ASSOCIATION AND OTHERS. Ecmascript language specification, 2011. 117

[12] ERLINGSSON, Ú., PEINADO, M., PETER, S., BUDIU, M., AND MAINAR-RUIZ, G. Fay: extensible distributed tracing from kernels to clusters. *ACM Transactions on Computer Systems (TOCS) 30*, 4 (2012), 13. 132

[13] FELDERMAN, B. Personal communication, June 2015. Google. 2

[14] FISCHER, K., YANG, S., MOK, B., MAHESHWARI, R., SIRKIN, D., AND JU, W. Initiating interactions and negotiating approach: a robotic trash can in the field. In *2015 AAAI Spring Symposium Series* (2015). vi

[15] FITZPATRICK, B., ET AL. memcached: a Distributed Memory Object Caching System. `http://www.memcached.org/`, Jan. 2011. 87, 96

[16] FONSECA, R., PORTER, G., KATZ, R. H., SHENKER, S., AND STOICA, I. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation* (2007), USENIX Association, pp. 20–20. 131

[17] GAILLY, J.-L., AND ADLER, M. GNU Gzip. `http://www.gzip.org`. 84, 89, 90

[18] GEEKSFORGEEKS. External Sorting. `https://www.geeksforgeeks.org/external-sorting/`. 67

[19] GNU COMMUNITY. 6 Line Control. `https://gcc.gnu.org/onlinedocs/cpp/Line-Control.html`, 2002. 32

[20] GNU COMMUNITY. 6.8 128-bit Integers. `https://gcc.gnu.org/onlinedocs/gcc-5.3.0/gcc/_005f_005fint128.html`, 2002. 86

[21] GOOGLE. glog: Google Logging Module. `https://github.com/google/glog`. 3, 95, 120

[22] GOOGLE. gRPC: A high performance, open-source universal RPC framework. `http://www.grpc.io`. 127

[23] GOOGLE. Snappy, a fast compressor/decompressor. `https://github.com/google/snappy`. 89, 90

[24] GREGG, B. Linux BPF Superpowers. http://www.brendangregg.com/blog/2016-03-05/linux-bpf-superpowers.html, 2016. 132

[25] GREGG, B., AND MAURO, J. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD.* Prentice Hall Professional, 2011. 132

[26] HOLLINGSWORTH, J. K., MILLER, B. P., AND CARGILLE, J. Dynamic program Instrumentation for Scalable Performance Tools. In *Scalable High-Performance Computing Conference, 1994., Proceedings of the* (1994), IEEE, pp. 841–850. 132

[27] HUANG, P., GUO, C., ZHOU, L., LORCH, J. R., DANG, Y., CHINTALAPATI, M., AND YAO, R. Gray failure: The achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (2017), ACM, pp. 150–155. 1

[28] HUDEK, T., BAZAN, N., GOLDEN, B., AND VARMA, S. WPP Software Tracing. https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/wpp-software-tracing, 2007. 95, 129

[29] INTEL CORPORATION. Using the rdtsc instruction for performance monitoring. *Techn. Ber., tech. rep., Intel Coorporation* (1997), 22. 50, 54, 110, 111, 125, 130

[30] INTEL CORPORATION. Intel 64 and IA-32 Architectures Optimization Reference Manual. *Intel Corporation, June* (2016). 92

[31] JAKOB STOKLUND OLESEN, S. Y. Varint, variable-length integer encodings. https://github.com/syang0/varint. 89

[32] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization* (2004), IEEE Computer Society, p. 75. 116

[33] LIM, H., HAN, D., ANDERSEN, D. G., AND KAMINSKY, M. MICA: A holistic approach to fast in-memory key-value storage. USENIX. 2

[34] The Linux Kernel Organization. https://www.kernel.org/nonprofit.html, May 2018. 87, 96

[35] LMAX Disruptor: High Performance Inter-Thread Messaging Library. `http://lmax-exchange.github.io/disruptor/`. 95

[36] MACE, J., ROELKE, R., AND FONSECA, R. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 378–393. 132

[37] MELMAN, G. spdlog: A Super fast C++ logging library. `https://github.com/gabime/spdlog`. 3, 95

[38] MOK, B., YANG, S., SIRKIN, D., AND JU, W. Empathy: interactions with emotive robotic drawers. In *2014 9th ACM/IEEE International Conference on Human-Robot Interaction (HRI)* (2014), IEEE, pp. 250–251. vi

[39] MOK, B. K.-J., YANG, S., SIRKIN, D., AND JU, W. A place for every tool and every tool in its place: Performing collaborative tasks with interactive robotic drawers. In *2015 24th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)* (2015), IEEE, pp. 700–706. vi

[40] MORTORAY, E. Wait-free queueing and ultra-low latency logging. `https://mortoray.com/2014/05/29/wait-free-queueing-and-ultra-low-latency-logging/`, 2014. 125, 130

[41] NAGARAJ, K., KILLIAN, C., AND NEVILLE, J. Structured Comparative Analysis of Systems Logs to Diagnose Performance Problems. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (2012), USENIX Association, pp. 26–26. 132

[42] NORVIG, P. *Natural Language Corpus Data: Beautiful Data*, 2011 (accessed January 3, 2018). 91

[43] OLINER, A., GANAPATHI, A., AND XU, W. Advances and challenges in log analysis. *Communications of the ACM 55*, 2 (2012), 55–61. 132

[44] OTT, D. Personal communication, June 2015. VMWare. 2

[45] OUSTERHOUT, J., GOPALAN, A., GUPTA, A., KEJRIWAL, A., LEE, C., MONTAZERI, B., ONGARO, D., PARK, S. J., QIN, H., ROSENBLUM, M., ET AL. The RAMCloud Storage System. *ACM Transactions on Computer Systems (TOCS) 33*, 3 (2015), 7. 2, 87, 95, 96, 106, 107

[46] PAOLONI, G. How to benchmark code execution times on Intel IA-32 and IA-64 instruction set architectures. *Intel Corporation, September 123* (2010). 14, 55

[47] PARK, I., AND BUCH, R. Improve Debugging And Performance Tuning With ETW. *MSDN Magazine*, April 2007 (2007). 3, 95, 124, 129

[48] PEKHIMENKO, G., GUO, C., JEON, M., HUANG, P., AND ZHOU, L. TerseCades: Efficient Data Compression in Stream Processing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 307–320. 85

[49] Performance Utilities. `https://github.com/PlatformLab/PerfUtils`. 5, 125, 126

[50] SANFILIPPO, S. Redis. `http://redis.io`. 2

[51] SIGELMAN, B. H., BARROSO, L. A., BURROWS, M., STEPHENSON, P., PLAKAL, M., BEAVER, D., JASPAN, S., AND SHANBHAG, C. Dapper, a large-scale distributed systems tracing infrastructure. Tech. rep., Technical report, Google, 2010. 131

[52] SIRKIN, D., MOK, B., YANG, S., AND JU, W. Mechanical ottoman: how robotic furniture offers and withdraws support. In *Proceedings of the Tenth Annual ACM/IEEE International Conference on Human-Robot Interaction* (2015), ACM, pp. 11–18. vi

[53] SIRKIN, D., MOK, B., YANG, S., MAHESHWARI, R., AND JU, W. Improving design thinking through collaborative improvisation. In *Design Thinking Research*. Springer, 2016, pp. 93–108. vi

[54] SITES, R. L. Benchmarking" Hello, World! *Benchmarking 16*, 5 (2018). 124, 125, 126, 130

[55] SLEE, M., AGARWAL, A., AND KWIATKOWSKI, M. Thrift: Scalable cross-language services implementation. *Facebook White Paper 5*, 8 (2007). 127

[56] SONITAAAAA. SohWhy9.png. `https://commons.wikimedia.org/wiki/File:SohWhy9.png`. vii

[57] Splunk. `https://www.splunk.com`. 117, 118, 132

[58] SQLITE CONSORTIUM. SQLite4 Variable-Length Integers. `https://sqlite.org/src4/doc/trunk/www/varint.wiki`. 85

[59] STALLMAN, R. M., MCGRATH, R., AND SMITH, P. *GNU Make: A Program for Directed Compilation*. Free software foundation, 2002. 16, 21

[60] THE APACHE SOFTWARE FOUNDATION. Apache HTrace: A tracing framework for use with distributed systems. `http://htrace.incubator.apache.org`. 131

[61] THE APACHE SOFTWARE FOUNDATION. Apache HTTP Server Project. `http://httpd.apache.org`. 87, 96

[62] THE APACHE SOFTWARE FOUNDATION. Apache Log4j 2. `https://logging.apache.org/log4j/log4j-2.3/manual/async.html`. 3, 95

[63] THE APACHE SOFTWARE FOUNDATION. Apache Spark. `https://spark.apache.org`. 87, 96

[64] THE APACHE SOFTWARE FOUNDATION. Log4j2 Location Information. `https://logging.apache.org/log4j/2.x/manual/layouts.html#LocationInformation`. 95

[65] VARDA, K. Protocol buffers: Googles data interchange format. *Google Open Source Blog, Available at least as early as Jul* (2008). 127

[66] VARDA, K. Capn proto: Encoding Spec. `https://capnproto.org/encoding.html`, 2013. 127

[67] VARDA, K. Capn proto: Introduction. `https://capnproto.org/index.html`, 2013. 127

[68] VARDA, K. Capn proto: Schema Language. `https://capnproto.org/language.html`, 2013. 128

[69] YANG, S. Log Analyzer: A collection of scripts to statically analyze log statements in open-source software. `https://github.com/PlatformLab/Log-Analyzer`. 87, 96

[70] YANG, S. NanoLog: an extremely performant nanosecond scale logging system for C++ that exposes a simple printf-like API. `https://github.com/PlatformLab/NanoLog`. 2, 3, 10, 21, 26, 44, 68, 71, 94, 116, 120, 130, 135

[71] YANG, S. NanoLogCompression: Contains the benchmarks used to compare the NanoLog compression algorithm vs. zlib and snappy. `https://github.com/syang0/NanoLogCompression`. 91

[72] YANG, S., MOK, B. K.-J., SIRKIN, D., IVE, H. P., MAHESHWARI, R., FISCHER, K., AND JU, W. Experiences developing socially acceptable interactions for a robotic trash barrel. In *2015 24th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)* (2015), IEEE, pp. 277–284. vi

[73] YANG, S., PARK, S. J., AND OUSTERHOUT, J. NanoLog: A Nanosecond Scale Logging System. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 335–350. 102

[74] Twitter Zipkin. `http://zipkin.io`. 131